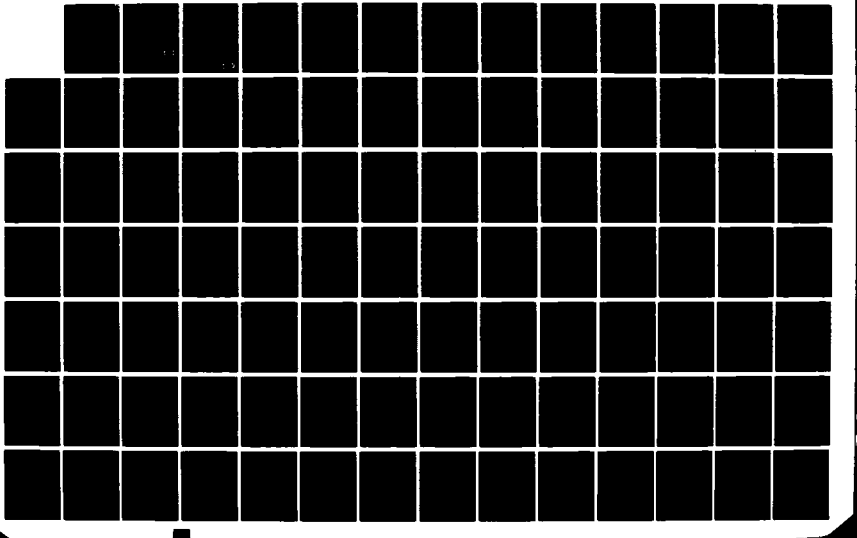
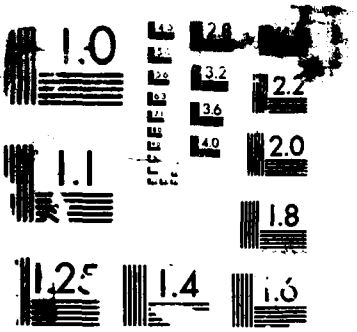
AD-A189 541 RAPID PROTOTYPING OF APPLICATION SPECIFIC PROCESSORS (U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING M GALLAGHER DEC 87 1/2

UNCLASSIFIED AFIT/GE/ENG/87D-19 F/G 12/6 NL





(1)

RAPID PROTOTYPING OF APPLICATION SPECIFIC PROCESSORS

THESIS

David M. Gallagher
Captain, USAF
AFIT/GE/ENG/87D-19

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/87D-19

RAPID PROTOTYPING OF APPLICATION SPECIFIC PROCESSORS

THESIS

David M. Gallagher
Captain, USAF
AFIT/GE/ENG/87D-19

AFIT/GE/ENG/87D-19

RAPID PROTOTYPING
OF
APPLICATION SPECIFIC PROCESSORS

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Electrical Engineering

David M. Gallagher, B.S.

Captain, USAF

December 1987

ACKNOWLEDGEMENTS

I would like to thank my wonderful wife Kathy for her loving support during this thesis effort. Her patience, cheerfulness, and love provided strength and stability to my home environment, allowing me to apply unburdened attention to my research.

I would like to thank my thesis advisor, Dr. Richard Linderman, for his help and wise insight. In addition to the outstanding wealth of knowledge he provides, he has developed an excellent VLSI design environment for thesis research. I appreciate the time and guidance Major Joseph DeGroat and Captain Bruce George provided through their careful reading of this manuscript. I would also like to thank the other students in the VLSI design group for the ideas and suggestions they have shared.

Most importantly, I would like to thank and praise the Lord Jesus Christ for the strength and wisdom He provides. The success of this thesis is due solely to His undeserved favor. To God be all the glory!

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

Chapter 4: VLSI Architectural Design

Chapter 5: VLSI Implementation

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF FIGURES (continued)

LIST OF TABLES

ABSTRACT

Numerous applications throughout the Department of Defense, industry, and academia require the design of custom processor architectures. Design of these processors, however, is normally a lengthy process. This thesis defines a methodology for rapid prototyping custom VLSI processor architectures. Using this methodology, the design and implementation of application specific processors can be reduced from several years to two months. This reduction makes a high-performance VLSI solution feasible for Department of Defense applications that would otherwise have settled for a lower-performance alternative.

The rapid prototyping methodology is based upon the specification of a general purpose architecture customized via microcode to solve unique applications. Since processing requirements will vary, the designer chooses appropriate macrocells from a design library to provide the best hardware support. A high-level language description of the problem is then translated into microcode. The microcode is automatically assembled and designed into a ROM (read-only memory), resulting in a processor customized to solve the given application. By allowing the designer to quickly convert ideas into implementations, the rapid prototyping methodology frees the designer to be creative rather than becoming mired in implementation details.

A general purpose VLSI architecture was designed to support the rapid prototyping methodology. The control section of the architecture centers on the microcode ROM (read-only memory) and a microcode sequencer, which provides proper addressing to the

ROM. The datapath section (I/O path, registers, and arithmetic hardware) uses the control signals from the ROM to perform the required processing. The datapath macrocells were designed in a "bit-slice" fashion, allowing easy configuration to different data types and widths. A prototype processor was implemented to test the architecture for functionality, performance, and operating characteristics. Additionally, a parallel floating point multiplier, applying Booth's modified algorithm in a Wallace tree structure, was fabricated. To further support the rapid prototyping methodology, several design tools were developed. These include a program to automatically generate a ROM in the Magic format and an extraction tool which generates a VHDL description of a circuit from a transistor listing, allowing high-level simulation of the circuit and thereby closing the "design loop."

The rapid prototyping methodology has been successfully applied to three different applications. These applications demonstrate that a custom application specific processor can be designed in less than two months using this methodology. This reduced design time also translates into reduced cost and program "risk." This dramatic decrease in design time could result in a significant increase in the usage of VLSI/VHSIC solutions to Department of Defense applications.

COMMON ABBREVIATIONS

Abbreviation	Explanation
ALU	Arithmetic Logic Unit
AND	"And" Gate
ASP	Application Specific Processor
CAD	Computer Aided Design
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal-Oxide-Semiconductor
DoD	Department of Defense
DRC	Design Rule Check
GND	Ground - "Low" Voltage
HOL	High-Order Language
LSB	Least Significant Bit
MSB	Most Significant Bit
MUX	Multiplexer
NaN	Not-a-Number
NAND	"Nand" gate
OR	"Or" Gate
PLA	Programmable Logic Array
PQ1	System Clock - Phase 1
PQ2	System Clock - Phase 2
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
Vdd	Source Voltage - "High" (+5v)
VHDL	VHSIC Hardware Description Language
VLSI	Very Large-scale Integration
VHSIC	Very High-speed Integrated Circuits
XOR	"Exclusive-or" Gate
XROM	Read-Only Memory w/ X-shaped Storage Cell

RAPID PROTOTYPING
OF
APPLICATION SPECIFIC PROCESSORS

CHAPTER 1

Introduction

1.1 Background

Advances in integrated circuit technology have made possible the design of VLSI (very large-scale integration) chips containing several hundred thousand transistors. A single chip can now perform functions that several years ago would have required a main-frame computer. As a result, VLSI architectures can now be applied to a broad range of difficult problems. The challenge for VLSI designers is to translate the ideas fostered by this new capability into reliable VLSI implementations.

An important VLSI area of study is the design of processor architectures to perform computation and provide control for special-purpose applications. Research programs throughout the Department of Defense (DoD) require application specific processors (ASPs) customized to accomplish one particular task. For example, several research efforts currently underway at AFIT have identified the need for VLSI architectures to provide system processing and control. These include:

1. AFIT CAM-puter system
2. WFT (Winograd Fourier Transform) project
3. VWE (Vector Wave Equation) processor
4. Silicon Brain and Optoelectronic Retina Architectures
5. Laser fusing for air-to-air missiles

To meet specifications, the VLSI processors/controllers for these projects must often be custom-designed. The specification, design, layout, verification, and fabrication of an application specific processor, however, can require two years or more to complete. Due to this lengthy time, and the perceived "risk" of using custom VLSI architectures, program managers are hesitant to commit to a VLSI solution to their processing needs.

One way to solve this problem is to design a general-purpose architecture that can be customized to specific applications. By customizing the microcode instructions stored in an on-chip ROM (read-only memory), a general-purpose architecture can be applied to specific problems. Using automated tools, the software microcode can be developed easily, allowing the rapid prototyping of a custom ASP. This ability to rapidly design reliable custom architectures will facilitate "VLSI insertion" into DoD programs. A primary purpose of this effort is to demonstrate that custom VLSI designs can provide reliable, economic, and higher performance solutions to many research programs which now use "off-the-shelf" products fabricated in old technologies.

1.2 Problem Statement

The goal of this thesis effort is to formulate a methodology for the rapid design, layout, and verification of custom processor architectures. The intention is to dramatically decrease the time required for these phases of processor development.

A general-purpose processor architecture containing a ROM to store microcode will be specified. A prototype processor will then be designed and implemented in VLSI. The

initial architecture will be implemented using a 3 micron CMOS, double-metal fabrication process. Throughout design, emphasis will be placed upon the ability to modify the designed the macrocells (via a "bit slice" design style) for use in different ASP applications.

1.3 Scope

This thesis is bounded both in the type of problems to which it applies and in the scope of the solution it presents.

1.3.1 Scope of the Problem. This methodology formulated for the rapid prototyping of ASPs is targeted to applications that are algorithmic in nature. The algorithm for the application should initially be specified by a HOL (high-order language) program. This program can then be translated into the microcode that will be incorporated into the ASP.

Applications that cannot be specified algorithmically, or which have special hardware requirements not easily provided by the general-purpose architecture, may not be good candidates for this rapid prototyping methodology. These types of problems will require increased design time and may not be easily implemented using existing macrocells. One example of a poor candidate for rapid prototyping is an I/O-bound problem requiring special bussing and external interface. Implementation would require extensive hardware re-design, lengthening the time required for prototyping. Another example is an application that uses trigonometric functions extensively. It would not achieve good performance without specialized hardware, thereby increasing its design time.

1.3.2 Scope of the Solution. The rapid prototyping methodology entails both the hardware and software aspects of ASP development. The hardware aspect of ASP development involves the design of the general-purpose architecture that can be customized with microcode to solve different problems. An important aspect of the hardware design and implementation is the creation of a cell library containing a variety of the macrocells necessary for ASP development. These macrocells can be easily modified, allowing the ASP architecture to be adapted to a variety of problems.

The software aspect of ASP development involves the specification of the microcode fields (instruction set), writing the microcode, and then incorporating this microcode into the design (creating the microcode ROM). A methodology is necessary for specifying the microword and translating the high-order language description into microcode. Automated tools are needed to assemble the microcode into binary form and then generate the ROM layout.

1.4 Summary of Current Knowledge

Considerable research has already been conducted at AFIT in the area of application specific processors. Several ASPs have been designed in past thesis efforts and class projects [Dia87, Gal86]. A prototype of the CAM system controller has been fabricated and tested [Fre86]. A processor for the WFT project has been also implemented and is currently being fabricated [She86].

In addition to the insight gained from these projects, numerous macrocells useful in an ASP architecture have already been designed. The CAM controller contains a control section similar to that envisioned for the ASP architecture. Adder, incrementable register stack, and barrel shifter macrocells are also available within the AFIT cell library

Although these cells will have to be adapted to the ASP architecture, the design of these cells is already well understood.

Perhaps the most significant work accomplished toward the rapid prototyping of ASPs is the development of the XROM Optimizer by Captain Rossbach [Ros85]. The XROM Optimizer inputs a file containing the desired contents of the XROM (a ROM whose basic storage cell is designed in an 'X' shape) and automatically generates an XROM layout in the Caesar format [Ous87]. The program also re-orders the address and data lines to minimize the number of transistors and drains required within the XROM. This, in turn, decreases the power requirements and increases the yield (and possibly the speed) of the design.

This XROM Optimizer greatly facilitates the rapid prototyping of ASPs. Once the general-purpose ASP architecture is designed, the major task in prototyping an ASP will be to customize the microcode stored in the XROM. Since the XROM Optimizer automatically implements the microcode in hardware, the only remaining task is to develop the binary description of the microcode for input to the XROM Optimizer.

1.5 Assumptions

In parallel with this effort, an IEEE-standard floating point adder is being developed and will be incorporated into the cell library. Design and initial implementation of this macrocell has begun and it should be available for use in future ASP architectures.

1.6 Approach

Since the software aspect of this effort was dependent upon the hardware developed, this thesis effort initially emphasized the design and implementation of the processor

architecture.

The first step in processor design was to analyze the types of problems that must be solved, and to survey the approaches to processor design that have been used by both industry and educational institutions. The requirements for the architecture were then specified and the processor designed. Once VLSI design was complete, a prototype ASP chip was fabricated.

After the processor architecture had been submitted for fabrication, the software methodology involved in rapid prototyping was addressed. Since the XROM Optimizer inputs a binary representation of the microcode fields, the desired algorithm must be converted to this binary representation. Extensions to the XROM Optimizer program were also examined to simplify its interface and to further develop its automated layout capabilities.

Once the fabricated chips were returned, they were be tested for both functional correctness and performance. These results in turn suggested improvements to the processor design that can be incorporated into future ASPs.

1.7 Materials and Equipment

The VLSI implementation of the processor architecture will rely heavily upon the Berkeley VLSI design tools [Cal86], including Magic, Mextra, and Esim. Additionally, design tools developed at AFIT such as the XROM Optimizer, Cstat [Lin85], and Stove [Gal86A] were used. These tools required use of both ELXSI computers, the VAX (SSC) computer, an AED767 graphics terminal with digitizer pad, a Versatec plotter, and a Sun workstation. This equipment was all available at the beginning of this effort.

Fabrication of the ASP required the support of the MOS Implementation System (MOSIS) located at the University of Southern California. Fabrication through MOSIS is funded by the Defense Advanced Research Projects Agency (DARPA) and required approximately 10 weeks. Communication with MOSIS was accomplished via the Arpanet system.

1.8 Sequence of Presentation

Chapter 1 has provided the background to this thesis effort and has defined the particular problem that will be solved. The scope of the problem and an approach to its solution were presented.

Chapter 2 reviews two general approaches that both industry and educational institutions have taken to processor design. Four specific architectures, representing both approaches, are overviewed.

Chapter 3 contains a detailed analysis of developing a methodology form rapid prototyping. Emphasis is placed upon specifying the processor architecture necessary for this methodology. Alternate approaches for developing an application specific processor are examined. The rationale for the chosen approach is then presented, followed by a the high-level architectural specification of the ASP processor.

Chapter 4 examines the detailed VLSI design of the ASP architecture which was described in Chapter 3. The architecture is divided into two major sections: 1) the control section, containing the microcode ROM and a microcode sequencer which provides addressing to the ROM; and 2) the datapath, which provides data storage and computation.

Chapter 5 describes the implementation of a prototype integer ASP. The procedure for "pad-to-pad" verification of the processor is presented, followed by a description of the fabrication process.

Chapter 6 details the rapid prototyping methodology and presents four test cases in which it has been successfully applied. The design time in each of these cases has been dramatically reduced.

Finally, Chapter 7 provides the conclusions from this effort and recommendations for further study in rapid prototyping of application specific processors.

CHAPTER 2

Review of Current Processor Design

2.1 Introduction

Considerable advances have been made in recent years in the design of general-purpose processor architectures. Architectures which fully support a 32-bit word have become commonplace. Although the need for even more powerful processors is universally acknowledged, debate still rages as to the best approach to processor design. This chapter examines the two popular design approaches and provides a survey of several specific architectures.

2.2 Processor Design Approaches

During the past few years, computer architects have begun realizing that "current architectures have serious shortcomings" [Mye82]. One critical problem has been termed the "semantic gap". Myers defined the semantic gap as "a measure of the difference between the concepts in high-level languages and the concepts in the underlying computer architecture" [Mye82]. Traditionally, hardware designers have made design decisions based on cost, ease of design, and performance of the machine instruction set. They have failed to analyze the nature of the high-level languages that use the architecture. As a result, the hardware often provides little support for many common high-level language functions. Myers identified arrays and procedure calls as two often used high-level constructs that generally receive little support from the underlying architecture.

A great deal of research effort is now being directed toward closing the semantic gap. VLSI designers are realizing that the real measure of a processor's performance is not how well it implements machine instructions, but in "the ability of the architecture to execute high-level language programs" [Hen84]. A processor that can be clocked at 25 MHz, but not providing adequate hardware support for many often used high-level language constructs, may show poor performance within an actual computer system.

Despite the large number of transistors that can be placed on a VLSI chip, the transistors must still be considered a limited resource [Kat85]. Efforts to reduce the semantic gap have diverged into two separate paths, relating to the use of this limited area. The major trend in processor design has been to use the increased transistor resources toward providing more complex instruction sets. These more complex instruction sets are designed to provide powerful machine-level instructions to support high-level language functions. A second approach to the use of transistor resources has been to implement a simple carefully-chosen instruction set and to provide hardware support to operate this simpler architecture at a higher speed than the complex architectures.

2.2.1 The CISC Approach. The "complex instruction set computer" (CISC) approach attempts to close the semantic gap by providing a more powerful instruction set. Many CISC processors are architectural successors of earlier processors [Kat85], such as the Intel 80386 and the Motorola 68020. These new processors provide a wider data path (number of bits of internal representation), a larger instruction set, and new addressing modes. As a result of the more complex instruction set, the instruction decoding logic of these processors must become more complex, often slowing down the maximum clock rate of the machine's entire instruction set [Col85].

A key feature of CISC architectures is the use of microcode. Microcode is a series of logic-level instructions stored on the processor chip, usually in a ROM. When the CISC processor receives an instruction, this instruction points to a sequence of logic instructions in the ROM that will be executed to accomplish the required task. Thus, the processor will usually require numerous clock cycles to execute a single command. The microcode allows the addition of more complex commands by merely adding new instructions to the microcode stored on the chip [Col85].

The complex instruction set has several advantages. Since a single complex instruction can accomplish more tasks, the size of the executable image can be smaller [Hen84]. This is significant, since the bandwidth (rate of information transfer) is much less between chips than it is on a chip [Kat85]. Thus, a CISC will pass fewer instructions between itself and memory, resulting in rapid execution. Another advantage of a complex instruction set is the ability to efficiently service a wider range of applications, since different applications use different sets of instructions [Hen84].

2.2.2 The RISC Approach. The "reduced instruction set computer" (RISC) approach to processor architecture attempts to optimize performance by providing a small instruction set that can operate at high speed. The basic premise of the RISC approach is that a program regularly uses only a small subset of a processor's instruction set. The majority of the complex instructions are rarely used. Proponents of the RISC approach feel that the inclusion of these rarely used instructions into a processor's instruction set "has more negative effects on performance than it has positive ones" [Kat85]. The complex control hardware required to implement these instructions slows down the speed of the processor, and the area required for this hardware could be better spent.

The common features that identify RISC architectures are [Col85, Joh87]:

1) Single-cycle execution. Since RISC processors use only a simple instruction set, most commands can be executed in a single clock cycle. Although many RISC architectures are pipelined and the latency through the pipeline may be several clock cycles, the execution stage of the pipeline is accomplished in one clock cycle. A new instruction can be inserted into the pipeline on during each clock cycle. This is in contrast to CISCs, in which one command triggers a sequence of microcoded instructions, requiring numerous clock cycles to execute.

2) Load / Store design. As already noted, more time is usually required to access information off-chip than to pass information within a chip. Since RISC processors wish to execute most instructions in a single clock cycle and to keep the length of the clock cycle to a minimum, they allow only simple load and store instructions to access memory. More complex instructions which must access memory, such as

$$\text{mem[C]} \leftarrow \text{mem[A]} + \text{mem[B]}$$

are not used since they would require either a longer clock cycle or multiple clock cycles to execute. The above operation would be executed by a RISC as a sequence of load and store instructions.

3) Hardwired control. An instruction loaded into the processor is decoded to directly provide the control signals that drive the hardware. This contrasts with the CISC approach, in which the incoming instruction provides a vector to a microcode routine stored on the processor. RISC processors generally do not use microcode.

4) Relatively few instructions and addressing modes. RISC processors take their name from the reduced number of instructions they support. By reducing the number of instructions and limiting the number of addressing modes, the control hardware is simplified and the clock cycle of the processor can be reduced.

5) Fixed instruction format. This simplifies instruction decoding, in turn reducing duration of the clock cycle.

6) More compile-time effort. RISC processors often reduce the hardware complexity in exchange for increased compiler time. This extra time is due to the post-compile processing required to ensure that the assembly code complies with hardware constraints. The time is paid for only once, during program compilation, and simplifies run-time execution. For example, several RISC architectures simplify their pipeline control hardware by placing constraints on the instruction flow the compiler is allowed to produce.

2.2.3 A Fuzzy Distinction. The "avalanche of publicity" [Col85] RISC processors have received in recent years has caused the distinction between RISC and CISC architectures to become somewhat clouded. A large number of new processors have been developed under the "RISC" label, but many of these do not satisfy the characteristics of a RISC offered above. Perhaps one source of confusion between the RISC and CISC approaches is the acronyms themselves [Col85]. They tend to promote the concept that the number of instructions in a processor's instruction set alone determines whether it is a RISC or CISC architecture.

The interpretation of performance comparisons has further clouded the issue of RISC versus CISC architectures. Benchmark tests on certain RISC processors have

shown that they operate significantly faster than their CISC counterparts. But it remains unclear if the increased performance of these processors is due to the RISC approach, the particular programs used in the testing, or the hardware advances (such as a register stack) incorporated into some RISC processors [Col85].

2.3 Relevant Processor Architectures

The special-purpose processors required for Department of Defense programs will not require many of the features necessary in general-purpose RISC or CISC processors. For example, custom processors do not need the memory management or operating system support that the latest general-purpose processors are providing. Much can be learned, however, from a study of RISC/CISC processor architectures. The implementation of the datapath in these architectures of particular interest. This section will provide an overview of four processor architectures: three that use the RISC approach and one using the traditional CISC approach.

2.3.1 IBM 801 Minicomputer. The IBM 801 project was the first architecture to use the RISC approach. The IBM 801 is not a VLSI microprocessor; rather, it is composed of a set of MSI (medium-scale integration) chips using ECL (a high-speed, high-power device technology). Even though it is categorized as a RISC, its instruction set contains over 100 instructions. To reduce the frequency of off-chip data fetches, a set of 32 registers is provided.

The IBM 801 is a true 32-bit architecture. Its internal data path operates on 32-bit two's-complement integers. It provides a 32-bit barrel shifter and logic functions. The ALU (arithmetic logic unit) provides only basic add/subtract functions, but special "multiply-step" and "divide-step" instructions provide an "add and shift" operation to

support multiplication and division routines. The primary data path requires only two busses, since the results from the ALU are not returned to the registers until the next clock cycle.

The majority of the 801's instructions execute in a single cycle. Testing of the 801 showed that it averages 1.1 cycles/instruction when tested on applications requiring few memory accesses.

2.3.2 The UCB RISC Project. The RISC project at Berkeley was the first RISC architecture implemented as a single VLSI chip. The Berkeley RISC project began in 1980 and the first chip, appropriately called RISC I, was fabricated in 1982. A follow-on chip, RISC II, was completed in 1983. Current research at Berkeley is concentrated in two areas: the development of a RISC-style architecture called SOAR (Smalltalk on a RISC) and a multiprocessor workstation called SPUR (Symbolic Processing using RISCs) [Rob87]. This paper will focus on the RISC II architecture, which is the foundation for Berkeley's current research.

RISC II is a 32-bit architecture, able to address up to 4 gigabytes of virtual memory. It performs 32-bit addition/ subtraction on signed two's complement numbers, but provides no hardware support for either multiply or divide operations. The ALU performs boolean logic and compare functions on 32-bit unsigned integers. The data path provides full-range shifting by use of a barrel shifter. RISC II uses a two-bus precharged data path, with the result of one cycle's operation returned to the register bank on the subsequent clock cycle.

RISC II is designed with a three-stage pipeline: fetch, operate, write result. Through the use of a 4-phase clock, the data path can both perform computations on

current cycle operands and write the previous cycle's results during the same clock cycle. Since the storage of results is delayed for one clock cycle, it is the compiler's job to rearrange instructions to ensure that the subsequent cycle does not depend upon current clock cycle results.

RISC II's instruction set is the result of an extensive evaluation of the type and frequency of instructions used by typical programs [Kat85]. It is streamlined, containing only 39 instructions, each only a single word long. The instructions are in a three-operand format and most can be executed in a single clock cycle (the *execution* requires one clock cycle, but the latency through the pipeline is three clock cycles). The only exception is the load and store instructions, which must access memory [Hen84].

The analysis of instruction usage also led the Berkeley team to realize that the majority of instructions involved either moves between operands or simple operations involving two sources and a destination operand. They concluded that for the processor to operate at a high rate of speed, the access to these operands had to be rapid. Since off-chip access time is significantly longer than on-chip access, the team concluded that the processor required a large amount of on-chip register storage. RISC II has a register bank of 138 registers, 32 of which are available at any time. The registers are organized as a register stack, so that subroutine calls and returns can be performed rapidly, without the overhead of storing the registers. Several of the registers overlap two levels of the stack, allowing parameter passing during subroutine calls. Proponents of the CISC approach credit this register stack (rather than the RISC approach) with the majority of the RISC II's performance success.

RISC II was built using a 3 micron nMOS process, and uses 41,000 transistors. It can be clocked at 12 MHz. Due to its simplified instruction set, it generally requires more

instructions than CISC architectures to accomplish a given task (approximately 50% more instructions than a VAX- 11). Since most of RISC II's instructions are single cycle. however, RISC II was able to run a series of 11 benchmark programs 2-3 times faster than its CISC counterparts [Hen84].

2.3.3 The Stanford MIPS Processor. Stanford University is currently researching the RISC approach with its MIPS project, which began in 1981. The MIPS chip was fabricated in 1983, using 4 micron nMOS technology. It uses 24,000 transistors and has a 4 MHz clock rate. MIPS uses a two-phase clocking scheme.

Like many of today's architectures, MIPS uses a 2-bus data path. Two different approaches were examined for reducing the propagation delay along the datapath. The first approach was a precharged bus structure. Testing showed that this structure would still have a 40ns delay, in addition to the precharge time [Hen82]. The second approach was a clamped bus structure. This structure reduces the voltage swing required to change the logic state of the bus by a factor of four. A voltage change on the bus is detected by a circuit similar to a sense amplifier. The bus delay with this setup was reduced to approximately 10ns and did not require precharging the bus.

The MIPS ALU provides full addition, subtraction, and logical functions, with a 80ns carry-lookahead tree adder. A barrel shifter provides shift capability. A special "add and double-shift" instruction supports Booth's modified algorithm for multiplication. An "add and single-shift" instruction supports division. No floating point support is provided. Internal storage is provided by 16 dynamic registers that are automatically refreshed if not written to.

The MIPS architecture employs a 5-stage pipeline: fetch, decode instruction, decode operand, execute, complete load. A new instruction enters the pipeline every two clock cycles, so that at any given time there may be three different instructions being executed by the pipeline. Since load/store hardware is separated from the datapath, MIPS is able to execute simple ALU operations during the same clock cycle that it performs a load/store instruction.

The MIPS acronym stands for "microprocessor without interlocked pipe stages" [Sil86]. A non-interlocked pipeline architecture provides no hardware support to eliminate pipeline conflicts. For example, a conflict would arise if the incoming instructions specified an "add" on a value fetched during the previous instruction. Since both instructions are in the pipeline at the same time, the "add" would require the value before the previous instruction had completed fetching it [Sil86]. Similar to the approach which other RISCs use, MIPS requires that the compiler prevent instruction conflicts such as this from occurring.

The MIPS compiler executes in two stages. The first stage is a standard compiler that converts the high-level language down to processor instructions. The second stage is a code re-organizer that re-orders the compiler output to avoid pipeline conflicts. The re-organizer also rearranges "branch" instructions to preclude flushing the pipeline (get rid of instructions that were pre-fetched but should not be executed because of the branch).

The MIPS processor was tested against the Motorola 68000 on such benchmark problems as Tower of Hanoi, Quicksort, and matrix multiplication. The MIPS processor out-performed the Motorola chip by a factor of five on these problems.

In 1986, Stanford fabricated MIPS-X, a 2 micron CMOS follow-on to the MIPS processor. Although the test chip had design errors, the Stanford team was able to test MIPS-X up to 17 MHz (the design goal was 20 MHz). The design errors were corrected and a new chip is currently being fabricated [Hen87].

2.3.4 Motorola 68020. The Motorola 68020 is a commercial microprocessor which uses the CISC approach. Built as a follow-on to the 68000, the 68020 [illegible] the first successful extension of a 16- bit processor into a 32-bit processor. It supports 32-bit address and data busses, as well as a 32-bit internal datapath.

The Motorola 68020 was implemented in 2 micron CMOS [illegible] 180,000 transistors and dissipating approximately 1.5 watts of power. It is clocked at 16 MHz, performing at a 2.7 MIPS (million instructions per second) rate [Mot86].

The 68020 employs a three-stage pipeline with a different instruction occupying each stage. The pipeline stages are: instruction decode, control generate, and execute. Instruction fetch is not included in the pipeline, since the processor possesses a 64-word (each 32 bits) instruction cache. Sixteen working registers are also provided. The 68020 instruction set is quite complex, with over 100 instructions and 14 different addressing modes. It is a superset of the 68000 instruction set.

Motorola provides a high degree of parallelism in the architecture, with three separate datapaths for instruction address, operand address, and data computations. Each of these datapaths provides a 32-bit adder. The data execution unit contains a barrel shifter and hardware support for Booth's modified algorithm. Parallelism is further provided by separating the bus controller section from the execution unit. This allows the bus controller to perform fetch/store operations during the same clock cycle

that computations are being performed in the ALU.

Although the 68020 contains a large microcode store, the average number of clock cycles to execute an instruction is only 7, compared to the 13.5 cycles required by the 68000. This efficiency, combined with the higher clock rate, allows the 68020 to execute instructions approximately 4.5 times faster than the 68000 [Mac86].

2.4 Conclusion

The four architectures surveyed represent only a small portion of the research that has been done both at educational institutions and by industry. They are representative, however, of the different approaches taken to processor design, and provide insight into the techniques that computer architects are using to increase processing speed.

Debate continues to rage between proponents of the two design approaches: RISC and CISC. Both RISC and CISC architectures have shown excellent performance. Although most commercial architectures still use the CISC approach, more and more are employing techniques developed for the RISC architectures, such as a large register stack.

The ASP architecture designed for this thesis effort does not require all of the features that general-purpose processors provide. For example, the ASP architecture does not need to support a variety of addressing modes, since it will be programmed at the microcode level. A great deal can be learned, however, by studying other VLSI designers' approaches to processor design. The simple instruction set employed by RISC architectures will reduce the clock cycle duration for the ASP architecture, without significant reduction in processing capability. Pipelining the ASP architecture will also serve to reduce the clock cycle. Parallelism, as provided by the Motorola 68020, can allow the ASP to perform more processing during each clock cycle. Each of the surveyed

processes provided similar computational hardware. A large register set, or even on-chip RAM, would allow the ASP to perform less memory access, increasing performance.

The ASP architecture will incorporate the best ideas from the processors surveyed in this chapter. Chapter 3 outlines the architectural decisions made in specifying the ASP architecture.

CHAPTER 3

Problem Analysis

3.1 Introduction

The need for application specific processors is widespread. Research efforts throughout the Department of Defense (DoD) require special-purpose processors and controllers to provide processing and control. This need provides the impetus for the development of a methodology to rapidly prototype special-purpose processors. Fundamental to this methodology is the definition of the architecture. The ASP architecture must be flexible enough for use in a variety of applications, yet powerful enough to provide high performance. This chapter analyzes the architectural design requirements and specifies a processor architecture to meet these requirements.

3.2 Possible Solutions

The architecture for a application specific processor can be provided in several ways. The special-purpose processor can be totally custom designed, as AFIT projects have often done in the past. A processor can be developed using gate arrays, with the assistance of an automatic layout tool. DoD efforts have largely opted for off-the-shelf processors, customized with an external memory store. A final solution is to use a semi-custom general-purpose architecture which can be customized to a particular problem via microcode. Each of these options will be examined individually.

3.2.1 Totally Custom Processors. When presented with an application requiring special-purpose processing, the most obvious solution is to employ a fully-custom VLSI architecture to solve the problem. By customizing the architecture to the application, the optimum performance can usually be achieved. The "area-time product" metric can be minimized through this approach.

The design time for the fully custom approach, however, is quite lengthy. In defense industry, this long delay from concept to tested silicon results in an hesitancy to use fully custom designs. Program managers cannot afford to wait the several years required to design and fabricate a custom chip. As a result, they usually elect to go with gate arrays or an off-the-shelf product, even though it may result in performance degradation.

In the academic environment, the required design time usually equates to 2-3 thesis cycles. A student is currently unable to take an idea from initial concept, through system design and VLSI implementation, and see it through to fabrication and testing. Not only does this hamper the student's learning, but also results in a loss of expertise when the student graduates, since whoever inherits the project must re-follow the same learning curve.

Another major difficulty with fully custom design is reliability. Many of defense industry's current projects have extremely high reliability requirements. For example, the Strategic Defense Initiative (SDI) effort understandably requires extremely high reliability. As the number of gates on a VLSI chip increases, the testing problem increases exponentially. Unless careful steps are taken during the design phase (design for testability), it can become impossible to achieve 100% fault coverage. As a result, it is extremely difficult to exhaustively test today's VLSI designs containing several hundred

thousand transistors. Their functional correctness is, therefore, difficult to verify. Additionally, even though a design may be functionally correct, the dependability of a VLSI chip is not established until the chip has been operationally tested over a period of time. As a result, defense industry has been slow to embrace custom VLSI designs. This issue of VLSI/VHSIC insertion is receiving increasing attention throughout the defense community.

3.2.2 Gate Arrays. Gate arrays are used widely by defense industry in the design of application specific integrated circuits (ASIC). Since the fabrication masks for these chips are already available and the fabrication process is well understood, the cost of implementing a design using gate arrays is probably much less than a custom approach [Wes85]. The regular structure of the gate array facilitates the use of computer-aided design (CAD) tools to perform automated layout of the design. Thus, the design of a circuit using gate arrays primarily involves the specification of the boolean logic of the circuit. The implementation phase, which normally would require the greatest investment in time, is virtually eliminated. The reliability of the design is high (assuming accurate CAD tools), since the underlying gate array has already been proven in other applications. Similar to custom design, however, providing 100% fault coverage is extremely difficult.

The primary drawback to gate arrays is performance. The designer is highly constrained in the types of circuit elements he has at his disposal. For example, to implement the boolean logic of "$a + bcd + d$", the designer would have to use a combination of several "and" and "or" gates. This might prove much less efficient than a direct implementation of the same expression using custom logic. The designer is also limited in the sizing of circuit elements using gate arrays. He cannot easily increase the current drive of

a circuit component or modify the ratioing. The result of these limitations is a design with a poor "area-time product". Although the increased area is of concern, the dramatic reduction in speed of the gate array approach, as opposed to a custom design, will likely result in unacceptable performance for high-speed applications.

3.2.3 Off-the-Shelf Components. In order to meet reliability specifications, most program engineers and managers within the defense industry are using older, proven circuits in their projects. Not only has the design itself been proven reliable, but the fabrication process is usually well established and better understood, resulting in higher yield. The cost of an off-the-shelf chip is usually low, since design costs are eliminated and the circuit is already being produced in mass quantities. Design time for the circuit is eliminated, and the overall "risk" to the project is greatly reduced.

Using this approach, a project requiring a special-purpose processor will generally take a general-purpose processor which is commercially available and then customize it to the particular application with an external memory store. The main problem with this approach is, again, the performance of the design, which is even worse than that of gate arrays. The lack of custom hardware support and the use of external memory will result in speed degradation of a factor of 10 or more over a fully custom design.

3.2.4 Semi-Custom Approach. As a compromise between a fully custom approach and the off-the-shelf approach, a semi-custom design can provide acceptable performance, cost, and reliability. In this approach, a general-purpose architecture is defined, and then customized to meet project specifications. Rather than customizing with an external memory store, the architecture is customized to the application via its internal microcode store. The remainder of the processor can be further modified to fit

the application by choosing different macrocells from a cell library.

The design time for this approach is minimal compared to the design time for a fully custom design. If the application requires no special hardware, the design merely involves the development of the microcode. Minor architectural changes can be accomplished rapidly by substituting macrocells from the cell library. All macrocells are designed to be easily modified, so that design time is minimized Using this approach, prototype design time can be reduced from several years to several months. The cost of this approach will be greater than the off-the-shelf approach, but much less than the custom approach due to the reduced time investment.

The reliability of the semi-custom approach approaches that of the off-the-shelf approach. Once the general-purpose architecture and the additional macrocells from the cell library have been proven reliable, changing the microcode personalization or adding cells from the library will cause little or no degradation in reliability. Design for testability is already built into the architecture, simplifying the generation of test vectors.

The performance of this approach, however, will be significantly better than the off-the-shelf approach for three reasons. First, the bandwidth from the on-chip microcode store is significantly higher than that of an off-chip memory. Transferring data within a chip involves moving the data over shorter distances and fighting less capacitance, resulting in faster data transfer and thus a higher clock rate. Additionally, storing the control software in a microcode store eliminates the need to decode the software as it is brought onto the chip. A normal instruction, which might be 8, 16, or 32 bits wide, must be decoded on the chip prior to being used. In contrast, a typical microcode word would be 60 bits wide, almost entirely decoded. Elimination of the decoding reduces the time to perform an instruction and allows an higher clock rate.

Secondly, a semi-custom approach allows for more parallelism in the hardware. If the software is stored off chip and decoded, the decoding restricts the means by which the hardware can be driven. Some parallelism which the hardware might provide is not allowed due to the limited instruction set available. Programming the architecture with microcode provides much more direct control over the hardware and will allow a greater exploitation of the hardware's parallelism.

Thirdly, the semi-custom approach allows modification of the hardware to the specific application. For instance, if the application is I/O bound, a second I/O channel can be added. If the application relies heavily on division, trigonometric functions, or some other operation which is not optimally supported by a general-purpose architecture, special hardware can be added to significantly increase performance. In order to keep design time short, performance of this approach will be less than the fully custom approach. A semi-custom architecture can be developed rapidly, however, usually with less than a factor of two degradation in performance.

The methodology for the rapid prototyping of ASPs will use the semi-custom architecture approach. This approach will result in a reliable product which can be rapidly designed, with a small decrease in performance from a fully custom design. The goal of rapid prototyping is not to produce a chip optimized for performance. Rather, prototyping is concerned with "proving the concept" and providing an approximation of the performance that can be achieved. This is significant, in light of the problems currently being encountered in VLSI/VHSIC insertion. A primary purpose of this effort is to show that a custom or semi-custom VLSI design can reliably and economically solve problems much faster than older, off-the-shelf products.

This approach is somewhat of a cross between the RISC and CISC approaches which were examined in Chapter 2. Like a CISC, the ASP architecture will rely on a microcode store. Instead of having a complex instruction set which vectors the processor into the microcode, however, the ASP architecture will normally have almost no external instruction set. In fact, many ASP applications may have a handshaking as simple as "Go - Done", where the host tells the processor to begin the processing and the processor reports when it has completed the predefined task. This "reduced" instruction set and the simpler decoding are more typical of the RISC approach.

3.3 Architectural Specification

The ASP architecture is designed using CMOS technology, which is the predominant technology in use today. As the number of transistors which can be placed on a chip increases, power dissipation has become a more and more significant issue. Figure 1 [Sed82] *shows a comparison of the speed and power requirements of popular technologies.* Its high power dissipation has made bipolar technology impractical for many VLSI applications. nMOS technology, which had been prevalent in VLSI design in recent years, suffers the problem of static power dissipation. As a result, CMOS technology has become more and more popular as chip complexity increases. Table 1 shows a comparison of CMOS to nMOS technology.

The microcoded ASP approach leads to several architectural decisions. The ASP must obviously include a ROM to store the microcode. The control section of the ASP is simplified from other architectures, since no instruction decoding is required. The primary architectural decisions deal with the specification of the datapath. The following sections describe the decisions which have been made.

Table 1. Comparison of CMOS and nMOS Technology [Wes85]

Characteristic	CMOS	nMOS
Logic Levels	Fully Restored	Weak Zero
Transition Times	Rise/Fall Equal	Rise Time Slower
Transmission Gates	Passes 0/1s Well	Passes Weak 1; T-gate Cannot Drive Another T-Gate
Power Dissipation	Almost No Static Some Dynamic Power Dissipation	Static Power to Output 0; Dynamic Power also
Power Supply	1.5 - 15 Volts	Fixed by Resistive Ratio
Density	2 Devices/Input to a Gate	1 Device/Input
Layout	Regular	Less Regular

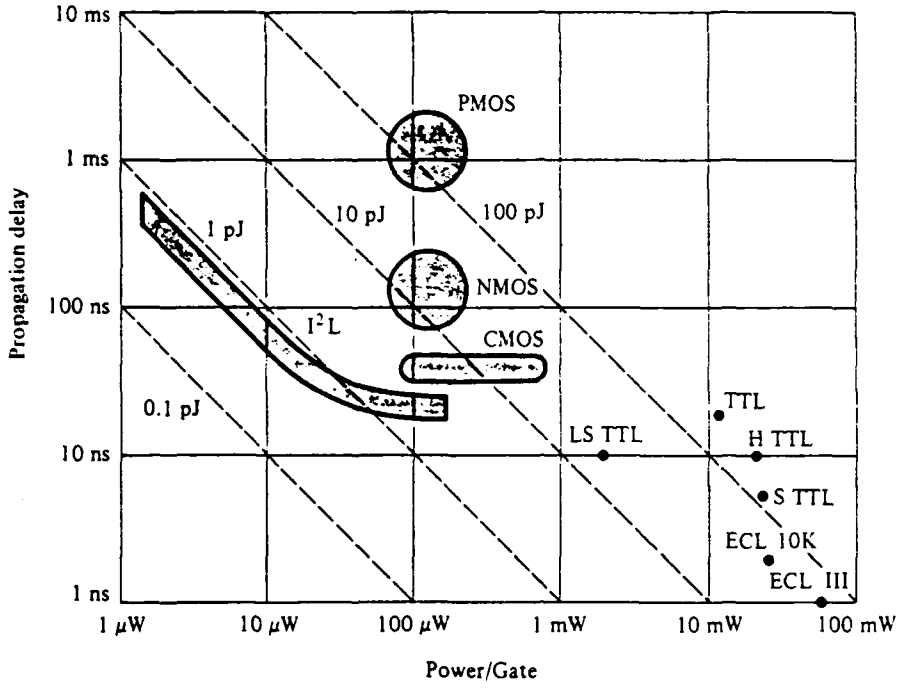


Figure 1 Comparison of Speed and Power Dissipation of Various Technologies [Sed82]

3.3.1 Number Representation. The fundamental question which must be addressed in datapath specification deals with the number representations which will be supported. Of particular significance, should the hardware support floating point numbers? The obvious answer is that the floating point representation is inherent in most scientific applications and must be supported. Some applications, however, will require only integer processing. One solution was to design two separate ASP architectures, one for floating point applications and another for integer-only applications. A floating point processor, however, must also support integer operations. Even if the data is floating point, integers are still used for addressing and in iterative constructs. Thus, an integer ALU must still be present on a floating point architecture. The integer-only

architecture, then, is actually a subset of the floating point chip rather than a separate chip. Only one ASP architecture is required, but it should be composed of a number a macrocells which can be easily added/deleted to provide the required hardware.

3.3.2 Width of Datapath. The ASP architecture will perform floating point operations on 32-bit IEEE Standard floating point numbers [IEE85]. This single-precision representation should suffice for the majority of applications requiring floating point computations. Applications which require integer operations, however, can vary in data width anywhere from a single bit to 64-bits. Even wider formats may require processing using a combination of hardware and software. The width of the ASP datapath must therefore be variable. The macrocells which compose the datapath should be built in a "bit splice" manner, so that the width of the datapath can be easily modified.

3.3.3 Number of Datapath Busses. Virtually all of the processors surveyed used either a two or three bus datapath. Processors using the three bus structure use two busses to provide data to the processing elements (ALU and shifter) and the third as a result bus to return data to the register array. Two bus architectures drive the data from the registers to the ALU/shifter on one clock cycle and then used one of these same busses to return the data to the registers during the next clock cycle or during a later phase of the same clock cycle. The architectures which used only two busses required either a complex clocking structure (4-phase clock) or imposed limitations upon the microinstructions in order to deconflict bus usage. The driving motivation for the two bus structure seems to be an effort to reduce the pitch of the datapath by eliminating the third bus. The ASP architecture will employ a three bus structure. This will simplify the design and eliminate restrictions upon the microcode.

3.3.4 I/O Path. Most applications for the ASP architecture should require a single I/O channel. The ASP should contain enough register space to store temporary data so that the I/O bandwidth requirement is limited. For some applications, however, a singe I/O channel will not suffice. For example, an application requiring the manipulation of large matrices would require a high I/O bandwidth. If the solution to the problem becomes I/O limited, the ASP architecture should be able to easily support a second I/O channel.

Each I/O channel will consist of two data streams: the address stream and the data stream. This is implemented in the ASP using a two register scheme. A memory address register outputs the required address to the memory. A bi-directional data register can either drive data to the memory or be loaded from memory. These two registers must also have access to the ALU for simple operations such as incrementing the address register.

3.3.5 General-Purpose Registers. Most processors which have been recently developed have either 16 or 32 general-purpose registers available. Even the RISC project at the University of California at Berkeley, which has a large register stack, has only 32 registers available at any one time. The ASP architecture should possess enough registers to minimize the I/O bandwidth, as discussed in the previous section. But the number of registers must be limited both for area and speed considerations. An increase in the number of registers results in an increased register decode time and in an increase in the length of the datapath, which increases datapath capacitance and slows execution.

The number of registers which the ASP will employ is highly dependent upon the application. Therefore, the register array must be built in a "bit-slice" manner, so that only the required number of registers are provided. A convenient number of registers is

2^{n-1} : 15, 31, etc. For example, if a 5-bit word is used to decode 31 registers, the other decoding can be used to specify none of the registers. This is necessary because other hardware, such as the I/O path, will need to use the datapath. Therefore, it is simpler to leave one decoding to specify that none of the general-purpose registers is selected.

3.3.6 Barrel Shifter. All of the architectures surveyed in Chapter 2 provided a shifter in the datapath. Since a full crossbar switch is impractical (requiring n^2 control signals for n bits), most processors provide a barrel shifter. There are a variety of design options available, however, for barrel shifters. A shifter can shift uni- or bi-directionally. The shifter can perform a circular shift or a circular shift through carry. Another option is to perform arithmetic shifts. In arithmetic shifts, left-shifts shift in zeros into the least significant bit. Right-shifts perform a sign extension on the most significant bit.

The type of barrel shifter used in the ASP will depend upon the computational requirements of the application algorithm. The design of a single shifter macrocell to perform all of the functions mentioned in the previous paragraph would be impractical. The methodology to rapidly prototype an ASP should therefore provide several library options for the types of shifters which might be required. The designer can then employ a barrel shifter from the cell library which is best suited to the application.

3.3.7 Arithmetic Logic Unit. The ALU must provide logical operations, integer arithmetic, floating point arithmetic, and possibly other operations such as trigonometric functions. Most design efforts either use software or employ a co-processing unit to provide the majority of the functions. The increased circuit density now available, however, allows these functions to be performed by on-chip hardware. As seen earlier, the on-chip bandwidth for data is much higher than the bandwidth between chips.

The ASP architecture will provide needed hardware on-chip. The key, then, is to determine what is "needed" hardware.

Obviously, addition, subtraction, and logic operations on integer numbers is easily and cheaply (in terms of area) provided. The functions will provide the ALU core for all ASP applications. What other hardware is provided will depend upon an evaluation of the application algorithm. The algorithm must be dissected to determine what types of operations are required and their frequency. Of special interest would be the "inner loop operations", which are performed iteratively during the algorithm. These operations which are used extensively should be supported with special-purpose hardware, while less often used functions might more wisely be provided by software. For floating point applications, a floating point adder macrocell is a necessity. Evaluation of the algorithm will determine if a floating point multiplier is a prudent investment of silicon. Similarly, an integer multiplier and hardware support for division should be employed only if warranted.

The ALU hardware for an ASP architecture will thus vary, dependent upon the algorithm. The ASP macrocell library, however, must provide support for the majority of processing requirements which might be encountered. As a minimum, macrocells to perform floating point addition and multiplication, and integer multiplication are required. Additionally, hardware to support division (both integer and floating point), trigonometric functions, square root, and exponentials should be available. Hardware in this second category might not actually perform the computation, but might support a software solution to the required operation. For instance, a small lookup table containing the first 4 bits of the quotient would significantly speed up a convergence routine used to perform the division function.

3.3.8 Literal Insertion. The majority of datapath computations are performed on data stored in the register set. Occasionally, data must be inserted from outside the datapath. The microcode should have the capability of injecting a literal into the datapath, so that constants can be stored in the microcode itself. The width of the datapath, often 32 bits or wider, precludes inserting all bits of the datapath during a single clock cycle. This would require an extremely wide microword, which increases capacitance and reduces access time of the ROM. One alternative is to store only a portion of the constant in the microcode, loading a full constant into the datapath via a series of insertions and shifts. If less than the full wordlength is inserted, the remaining bits in the datapath must also be controlled. This can be accomplished using a single bit from the ROM, which determines if the remaining bits should be driven high or pulled low.

3.3.9 Control Section. Programming the ASP architecture at the microcode level significantly simplifies the control section, since decoding of off-chip instructions is not required. Since all control signals are generated by the microcode ROM, processor control is chiefly a problem of generating the proper addressing to the ROM. Figure 2 shows a block diagram of the required hardware.

The program counter provides the address to the ROM for the next microword instruction. The next address hardware is required to provide sequential microcode execution, branch capability (conditional or unconditional), subroutine call and return, and external address control. External address control provides the designer with the capability to directly load the next ROM address from an source outside the control section: possibly from an instruction mapping PLA [Fre86], the input pads, or the datapath.

In order to maximize the clock frequency of the processor, the length of the critical timing path must be minimized. During each clock cycle, the program counter must

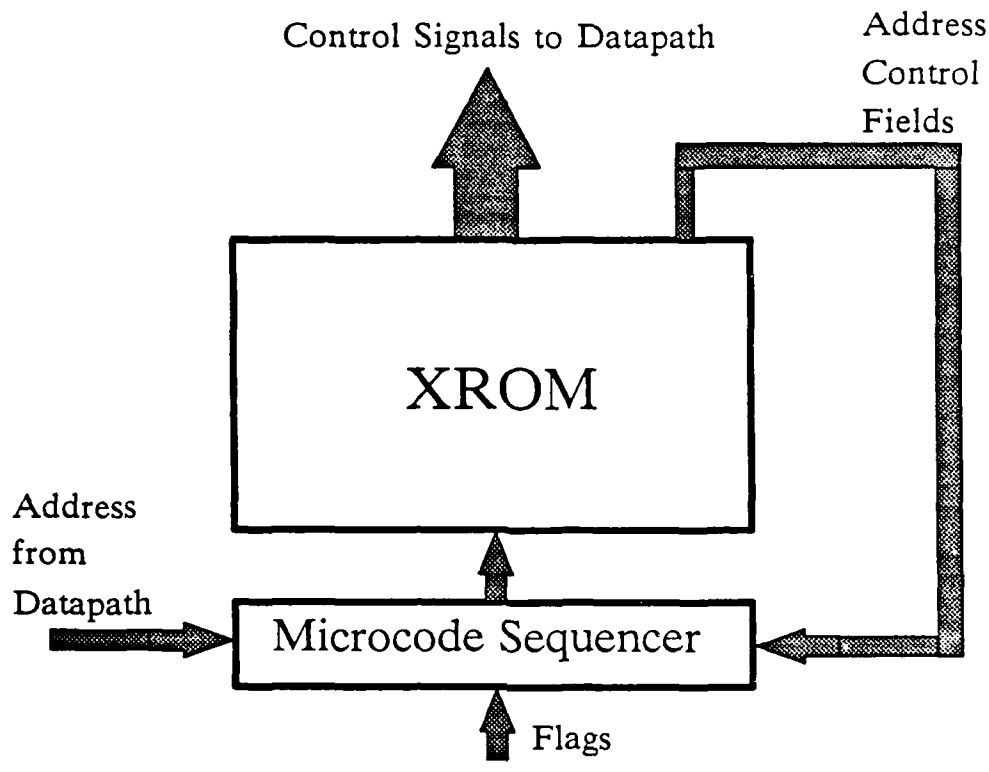


Figure 2. ASP Control Section

provide the address to the ROM, which, after its access time, will produce the required control signals. These signals must be driven to the datapath, at which time the actual processing can begin. The critical timing path through the chip is from program counter providing the correct addresses until all computation has been completed. This critical timing path can be broken, however, by the use of "pipelining". Using pipelining, the results of one stage of the pipeline is transferred to the next stage of the pipeline, to be operated upon during the subsequent clock cycle. When applied to the control section of the ASP architecture, the fetching of control signals from the ROM is accomplished on the clock cycle prior to these signals actually being used. The control signals from the

ROM are stored in a pipeline register until the next clock cycle. In this manner, the datapath is operating on control signals fetched during the preceding clock cycle, while the control section is simultaneously generating the next set of control signals. Figure 3 shows the ASP control section with the pipeline register added.

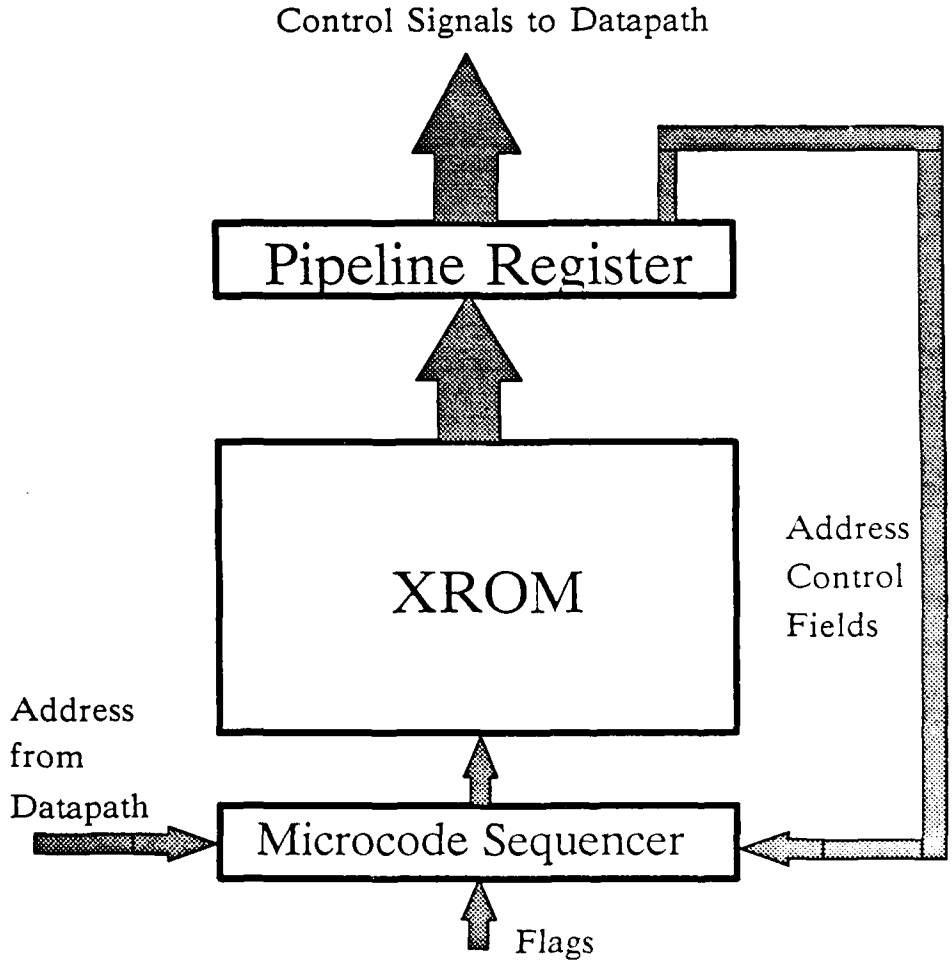


Figure 3. ASP Control Section with Pipeline Register

A pipelined control section has the following advantages:

1) It allows the majority of the clock cycle for operations specified by the control word. If a pipeline register is not present, operations cannot begin with the control word until it has been read from the ROM. This ROM access would require nearly as much time as the computations themselves.

With the pipelined approach, the word which was accessed from the ROM during the previous clock cycle is driven to the rest of the chip on the leading edge the new clock cycle (rising edge of PQ1). All required computations have from the rise of PQ1 until the data is latched back into the registers (on falling edge of PQ2) to complete. This allows all but 2-3 nanoseconds of the clock cycle for operations.

2) The pipelined approach removes components off of the critical timing path. Without a pipelined approach, other components cannot begin executing until after the ROM has been accessed. Thus, the length of the clock cycle is set by the access time of the ROM plus the time for the slowest other components to perform their functions. With the pipeline register, components such as the next address hardware, which might otherwise be on the critical path, are removed from the critical path. Thus, they can be designed simpler, smaller, and more reliably.

3) The pipelined approach provides buffering for the control signals coming out of the ROM. The AFIT XROM is designed to precharge during PQ1. During PQ1, all bit lines in the XROM are driven high. Since the outputs of the XROM may be inverted by sign bits, the control signals coming out of the XROM will be random in value during precharge. After PQ1 falls, the selected bit lines will begin to be pulled down and the outputs of the XROM will settle to their correct value. However,

during this access time the control signals are in a unknown state and should not be driven to the rest of the chip.

A pipeline register solves this problem. It latches the value of the XROM outputs on the falling edge of PQ2, after these outputs are valid. During the next clock cycle, these valid outputs are driven to the chip. The lack of buffering on previous VLSI designs at AFIT has resulted in problems during testing of the fabricated chips [Fre86].

4) A pipelined control section allows for a higher clock rate. As already seen, the clock cycle for a non-pipelined architecture is determined by the ROM access time plus the time to perform the required operations. With a pipelined approach, the length of the clock cycle is the maximum of the ROM access time or the execution time of the control signals. For the ASP architecture, the ROM access time will normally be slightly shorter (dependent upon the size of the ROM) than the execution time of the datapath. Thus, the datapath execution time will impose the limiting factor upon the ASP architecture's clocking frequency.

5) The pipeline register can provide testability to the design. The pipeline register can be designed to act as a shift register during testing. In the test mode, the contents of the shift register can be shifted out to provide observability of the XROM outputs. Additionally, new control values can be shifted in, providing controllability to the design.

The use of a pipeline register is not without its disadvantages. The disadvantages and problems associated with the pipelined approach are highlighted below:

1) The pipeline register requires additional hardware. Although the hardware

required is fairly simple, the pipeline register requires the storage of each output bit of the ROM. Thus the pipeline register will run the entire length of the ROM above the sense amps, adding to the effective height of the ROM.

2) The pipeline register requires extra microcode support. During sequential microcode execution the only penalty imposed by the pipeline register is the extra clock cycle to fill the register. For non- sequential execution, however, the pipeline register requires special handling within the microcode.

When the microcode branch logic encounters a jump instruction, it is operating on control signals which were fetched from the ROM during the previous clock cycle. The ROM has already fetched the next sequential instruction, which would normally not be executed if a branch occurs. This problem can be handled in various ways.

The hardware approach would be to prevent execution of the next instruction by somehow clearing the pipeline register during the next clock cycle, effectively "flushing" the pipeline each time a branch instruction is executed. This would require additional hardware, and, probably more importantly, would waste a clock cycle each time a jump is required.

A better approach is to handle the branching problem with the microcode. If the instruction following the branch is a NOP or some "don't care" instruction, this instruction can be executed without causing any side effects. Better yet, the branch instruction can be put into the microcode one instruction prior to the point where the designer wishes for the branch to occur. Thus, the instruction which follows the branch will be executed "on the fly" since it will already be in the pipeline, thus producing no wasted clock cycles. Studies at Stanford [Hen84] have shown that micro-

code can be manipulated such that the pipelined approach will typically result in less than a 5% increase in size of the code.

3) The pipeline register results in a delayed response to external control signals. If the ASP is looping, waiting for an external signal, the arrival of the external signal may occur during the execution of the loop instruction or during the "don't care" instruction which follows it. Thus, the ASP may require either one or two clock cycles to begin its response to the external signal.

The ASP architecture will employ a pipelined control section.

3.4 Conclusion

This chapter has provided an analysis of the architectural requirements of the ASP architecture and examined various solutions to the problem. A semi-custom approach, which is customized by microcode, was chosen as the proper approach for the ASP architecture. The architecture of the ASP was then specified at the macrocell level. This architecture can be easily modified to solve a wide range of algorithmic-type applications. The next chapter will describe the VLSI design of the macrocells required to meet the given specifications.

CHAPTER 4

VLSI Architectural Design

4.1 Introduction

This chapter describes the VLSI design of the macrocells for the application specific processor architecture. As described in the previous chapter, the actual hardware that will be employed for a particular application is dependent upon the problem algorithm. Therefore, the VLSI design of the ASP architecture centers upon the design of a library of macrocells which can be easily assembled to provide the required hardware. The ASP architecture (Figure 4) is comprised of two major sections: the control section, which generates the necessary control signals, and the datapath, which performs the data processing. Due to the ASP architecture's pipelined control section, the datapath operates in response to control signals fetched during the previous clock cycle. This chapter describes the VLSI design of the control section and datapath.

4.2 ASP Control Section

The control section of the ASP architecture is responsible for generating the control signals to drive the datapath computations. Unlike the control sections of most processors, which are primarily involved with decoding an instruction which is received from off-chip, the ASP control section derives its control signals directly from an on-chip ROM. Since the outputs of the ROM do not require decoding, the primary job of the ASP control section is to provide the correct addressing to the ROM so that the proper control signals are generated. Figure 5 shows the primary functional blocks of the ASP

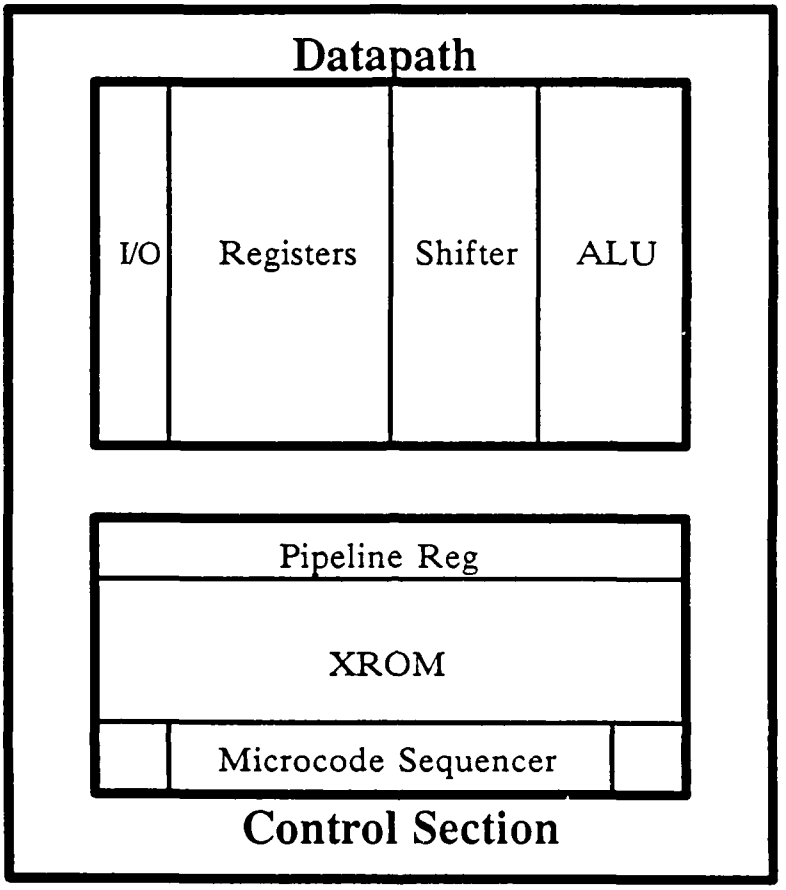


Figure 4. Major Sections of ASP Architecture

control section. Central is the AFIT XROM, which stores the control signals. The pipeline register, which receives the XROM's outputs, is responsible for delaying them for one clock cycle. The next address hardware, or microprogram sequencer, is responsible for providing the address to the XROM. Each of these macrocells will be examined in detail.

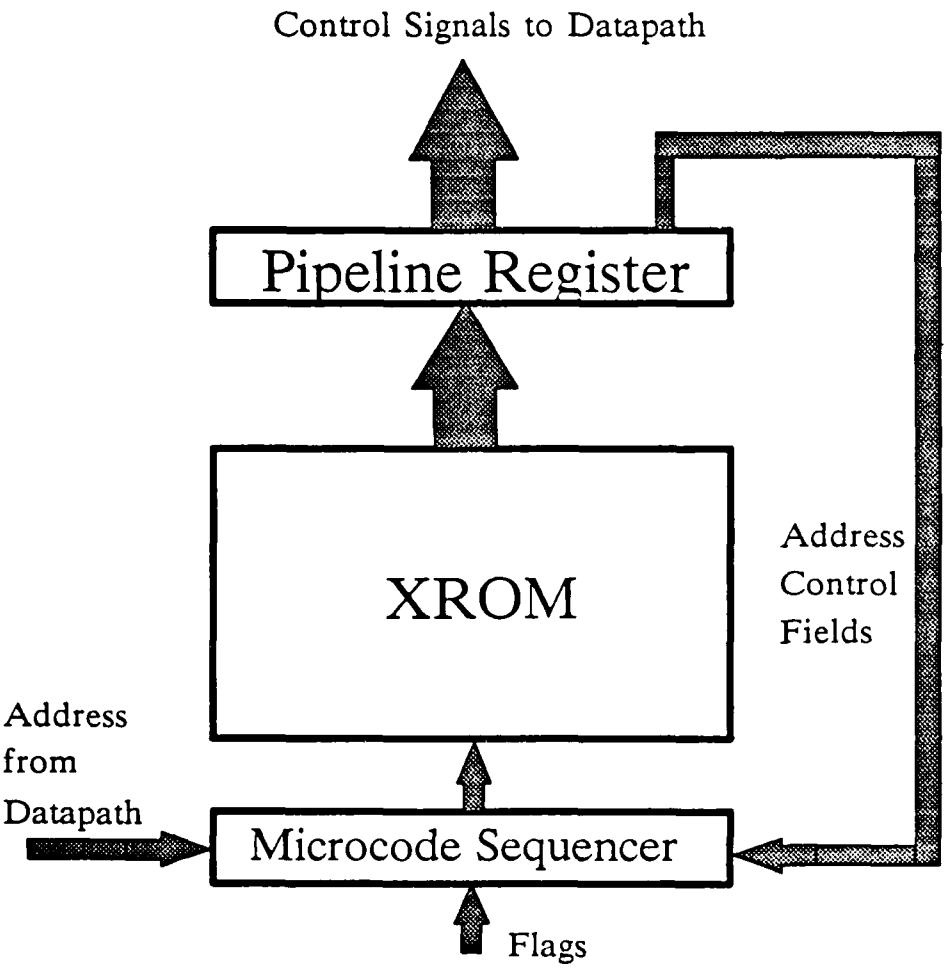


Figure 5. ASP Control Section

4.2.1 ASP XROM. The design for the AFIT XROM (Figure 6) was first presented by Captain Paul Rossbach [Ros85]. Since the XROM was designed to be used in high-speed processors, it was designed for access times of less than 50 ns. The XROM was also designed to facilitate automatic silicon compilation. The address decoders are responsible for decoding the input address and selecting one of the horizontal wordlines to go high. The presence or absence of transistors along this wordline determines

whether the corresponding output bit for the selected word will be high or low.

Figure 7 shows a close-up of the basic XROM storage cell. Note the "X- shape" from which the XROM derived its name. Depending upon A0, the least significant bit of the address, either the A0 or A0bar line will be pulled to ground and the other will be charged to 5 volts. When the selected wordline is raised to 5 volts, the gates of transistors along that wordline are activated, allowing the bitline to discharge through the transistor to either the A0 or A0bar line, depending upon which is set to ground. Thus, the presence of a transistor in a given location will allow the bitline to be pulled low, indicating that this bit is high for the particular address selected. Note that in the case in which transistors connect to both the A0 and A0bar line, "fighting" will occur on the bitline and it will settle to a voltage slightly less than 2.5 volts. This causes no difficulty, however, because the senseamp circuitry is designed to "sense" a low voltage for any voltage below 4 volts.

Captain Rossbach also developed the initial version of an optimizing silicon compiler for the XROM [Ros85, Ros87]. The compiler was later refined by Captain Linderman and this author. This tool provides automatic layout of the main XROM array. The optimizer inputs a file which contains an integer representation of the binary values that are to be stored in the XROM. Output of the optimizer is a set of 12 files which describe the XROM in a Caesar format.

In addition to automatic layout, the XROM Optimizer attempts to minimize the transistor count and the number of drains in the main storage arrays [Ros87]. This device minimization serves to increase reliability and to reduce power dissipation and access time. The primary vehicle for reducing the transistor count is the use of row and column sign bits. If any given word in the XROM contains more '1's than '0's, a row sign bit

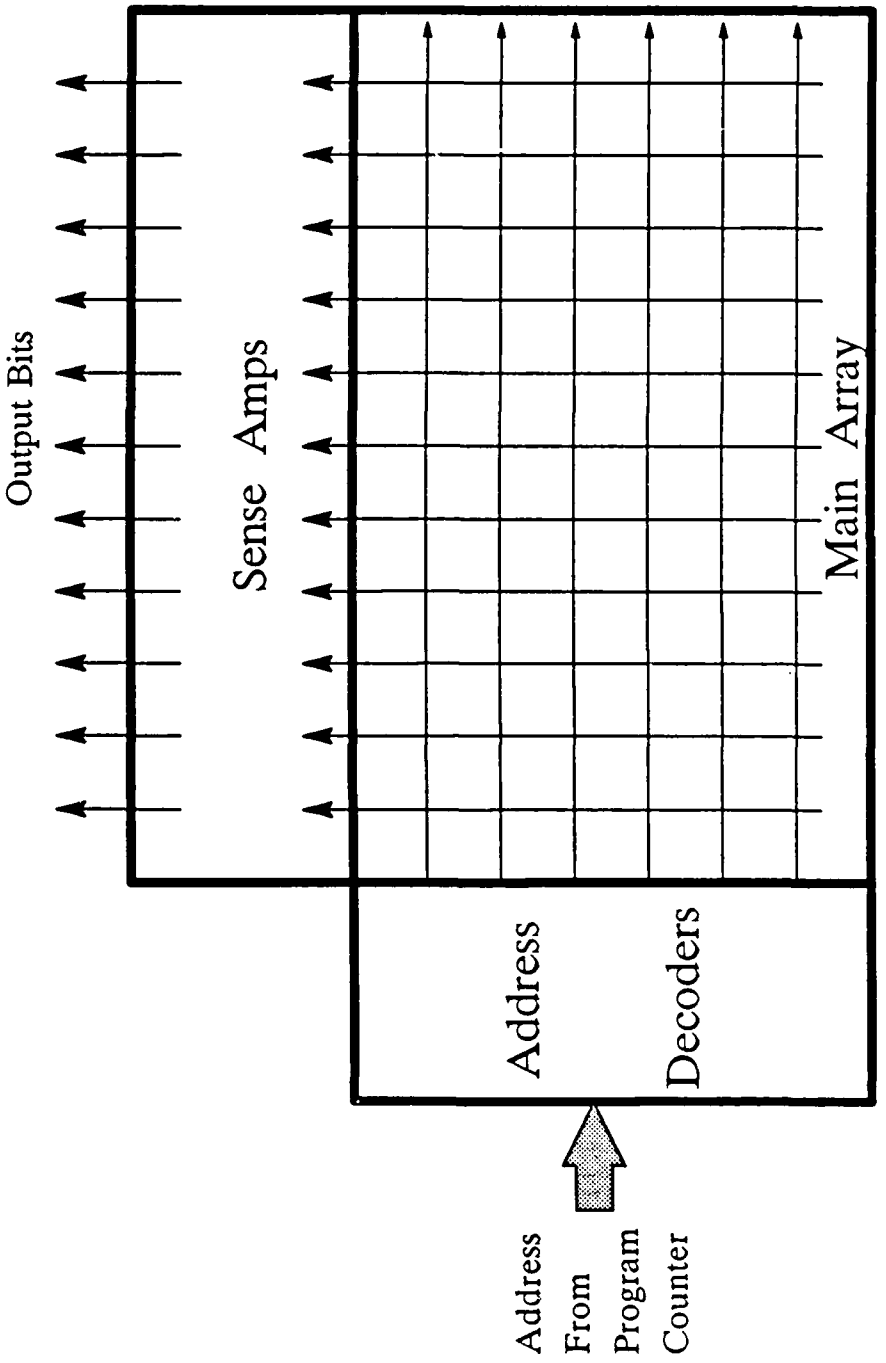


Figure 6. AFIT XROM

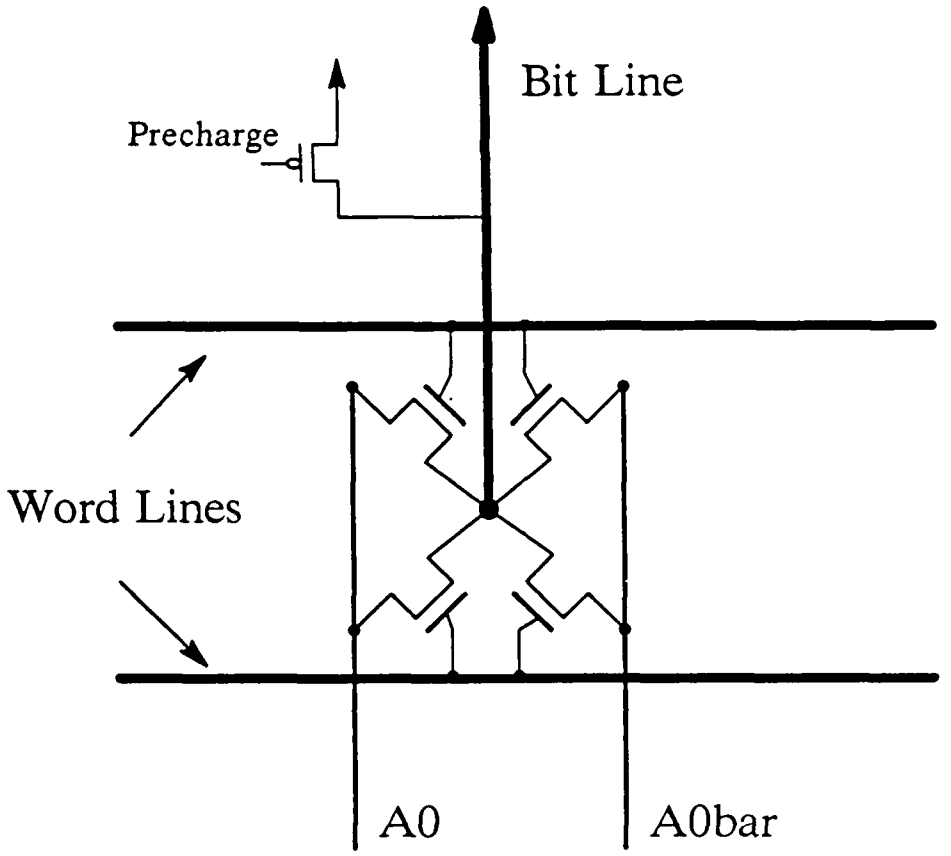


Figure 7. XROM Storage Cell

can be employed to indicate that the entire row is inverted. Thus, the polarity of the row is reversed and there will be less '1's than '0's. Likewise, column sign bits are used to ensure that all columns will contain less than 50% transistors. If none of the four transistors which form the "X" are present, the drain which forms the junction of these transistors can be removed. By reordering of rows and columns, the number of unneeded drains was maximized. In the case of a 54K XROM developed for the WFT16 effort, the optimizer was able to achieve a 44% reduction in transistor count and a 51% reduction

in number of drains [Ros87].

Since this tool was developed, AFIT and a majority of other institutions are now using the Magic layout tool for VLSI design, as opposed to Caesar. As part of this effort, the layout portion of the XROM Optimizer was modified to output the XROM description in the Magic format. Additionally, the layout program was extended, so that all required XROM subcells are now placed. Further extension is still possible, such that the pipeline register cells are placed and the control bus is automatically "personalized" to the particular column ordering.

To generate a custom XROM, the Optimizer must be run on the desired data to be stored. In addition to the Magic files produced by the Optimizer, cells from the XROM library must be included. Due to possible timestamp differences in the cells, Magic may want to design-rule check (DRC) the entire XROM. Additionally, Magic tends to be quite slow when handling a cell hierarchy as laid out by the Optimizer. Magic's performance can be significantly improved by flattening the cell hierarchy in the main XROM array.

For a further description of the AFIT XROM design, see [Ros85] and [Ros87].

4.2.2 ASP Pipeline Register. The pipeline register is designed to sit atop the XROM, buffering between the senseamp outputs and the remainder of the RISC chip. The current XROM design places the senseamps 52 lambda apart, setting the maximum width of the pipeline cell at 56 lambda (allowing for a 4 lambda overlap of power lines). The pipeline register then becomes simply an array of pipeline cells, one cell immediately above each of the XROM's senseamps (Figure 8).

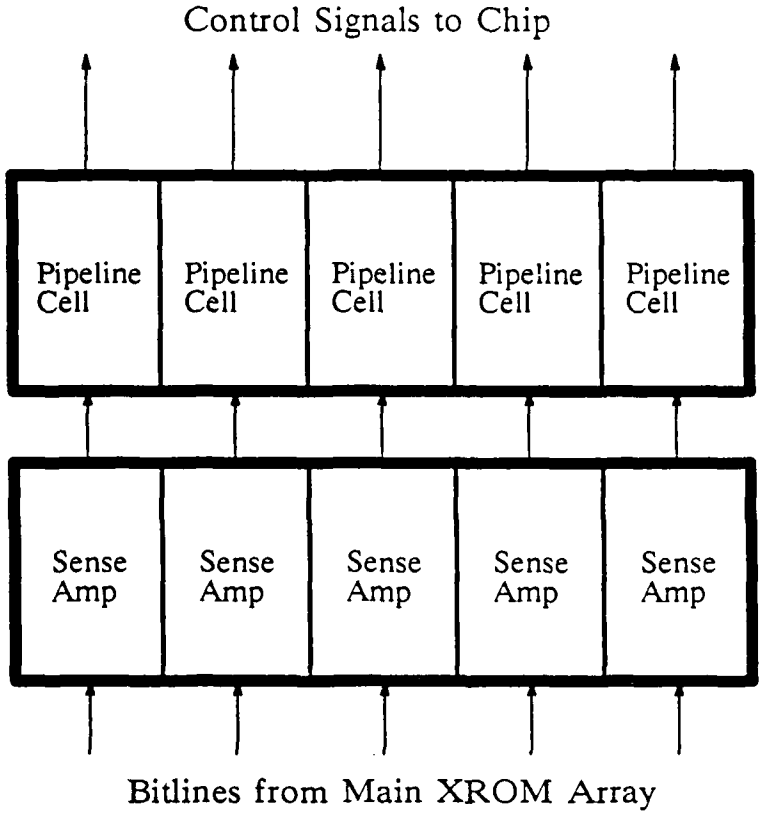


Figure 8. Pipeline Register

The design of the pipeline register cell is simple. The pipeline register must merely latch the control word while it is valid and then drive it to the rest of the chip on the next clock cycle. This can be accomplished using a master-slave flip-flop (MSFF), as shown in Figure 9. The output of the MSFF is staged up through inverters to provide the current drive to send the control signals across the chip. The initial design of the pipeline register called for resetting the register during chip reset. This, however, is not necessary and was not implemented. Resetting the register to 0 would not insure that all control signals were being deactivated, since some are active low. Additionally, even if

the register is reset, the XROM will fetch another invalid control word during the cycle after reset (during this bad clock cycle, the program counter will be reset to zero, and will provide valid signals from then on). Thus, resetting the pipeline register would be meaningless. The invalid control word will cause no problem to the chip, since during chip reset, the chip is storing no data which can be lost by the bad control word.

4.2.3 Microprogram Sequencer. The microprogram sequencer is responsible for providing addressing for the XROM control word. The sequencer hardware for the ASP architecture was adapted from a macrocell designed by Lt. French for the CAM RISC [Fre86]. The design is based on the microprogram sequencer described by Mano in [Man82].

Figure 10 shows a block diagram of the ASP microprogram sequencer. The design is centered around the program counter, which contains the actual address which will be driven to the XROM. The remainder of the functional blocks are responsible for loading the program counter with the proper address for correct program execution.

As shown in Figure 11, the next address to be loaded into the program counter can come from one of four locations:

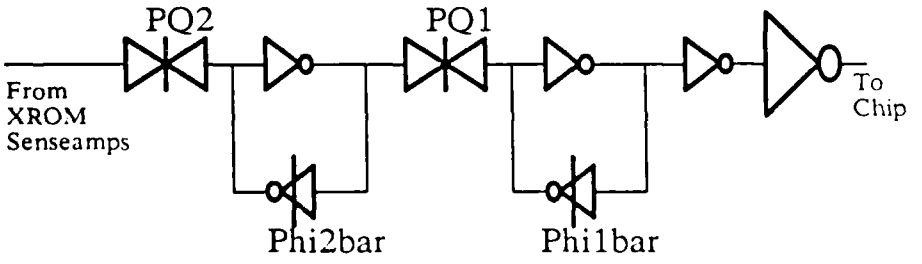


Figure 9. Pipeline Register Cell

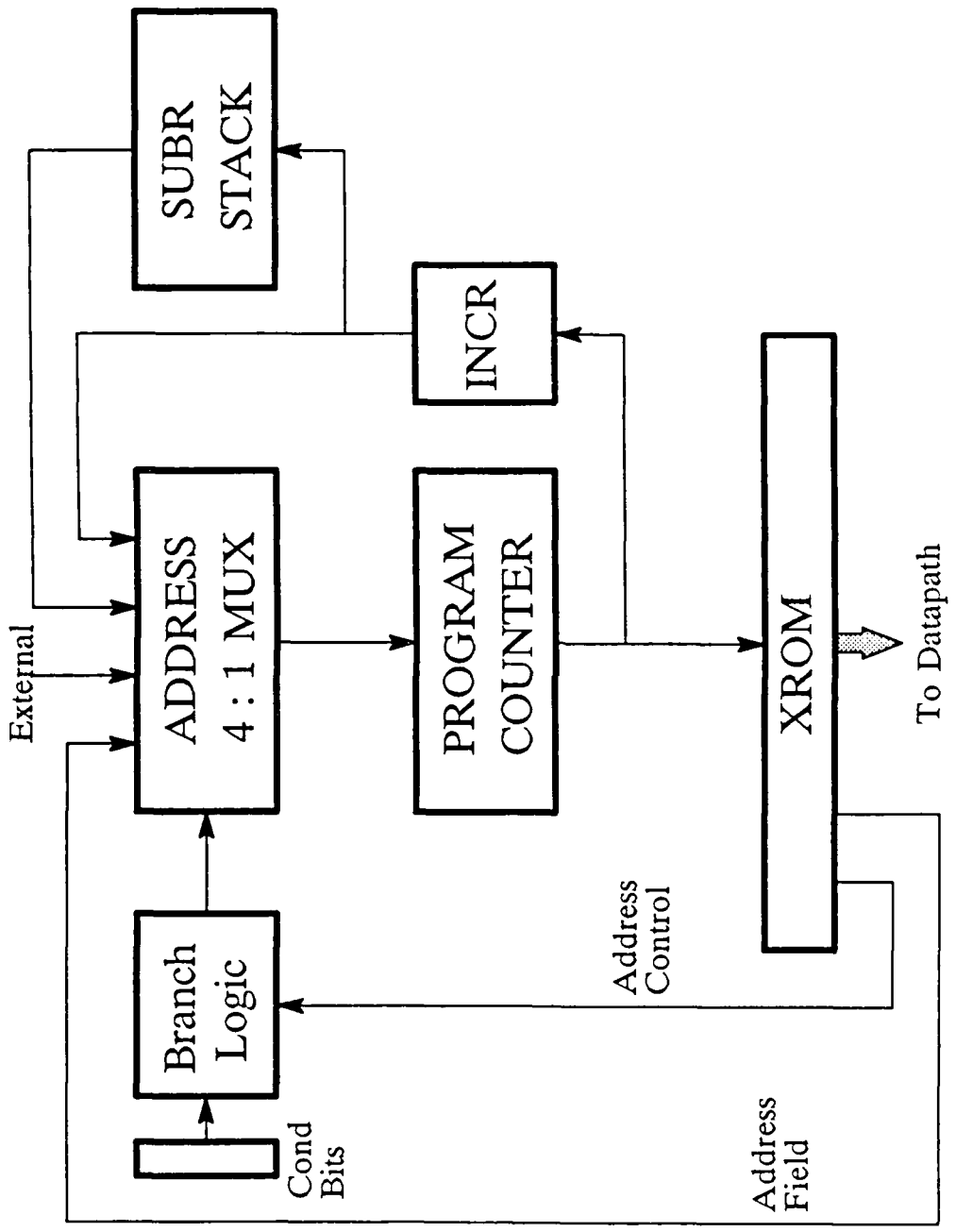


Figure 10. Microprogram Sequencer

1. The incremented contents of the program counter. Loading this value into the program counter provides sequential program execution.

2. The contents of the subroutine stack. This option equates to a return from subroutine.

3. The contents of the XROM Literal/Address field. Loading this field provides program branching capability.

4. The contents of the external address bus. Loading this value allows direct branching to a XROM memory location specified outside the control section. The external address bus can be connected to the datapath to allow branching to a location specified by a register or to the chips input pads, so that the system can directly control addressing of the XROM.

4.2.3.1 Micro-Program Counter. The micro-program counter provides addressing to the XROM decoders. The number of bits, or width, of the program counter (n) is determined by the number of words in the XROM (w), such that: $w \leq 2^n$. Therefore, an XROM with 1K words would require a 10-bit program counter. The design a single bit of the program counter is shown in Figure 12. The cell is basically a MSFF which can be loaded from the address multiplexer, the output of the incrementer, or be reset. The reset line is connected to chip reset, so that the program counter will start up with a deterministic value. Similar to the pipeline register seen earlier, the program counter drives on PQ1 and latches (loads) on PQ2. The microcode sequencer was designed so that the width of the program counter can be easily modified by merely adding or deleting the subcells which form the individual bits.

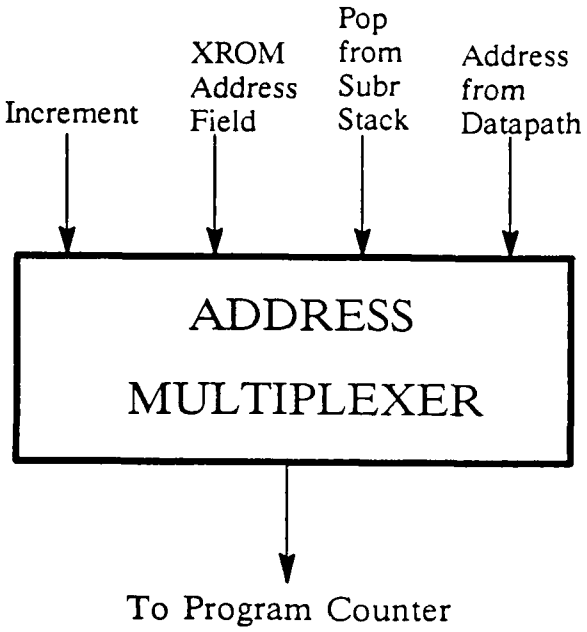


Figure 11. Sources of Sequencer Next Address

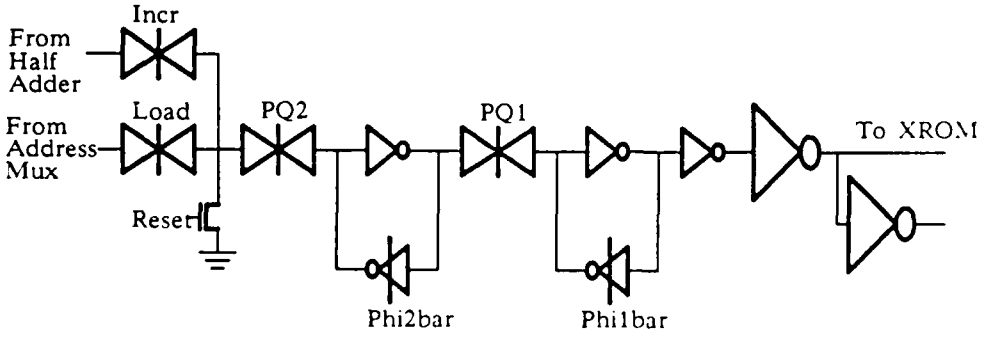


Figure 12. Program Counter Cell

4.2.3.2 Incrementer. Obviously, the incrementer is responsible for computing the value of the program counter plus one. Since the equation for an incrementer is:

Sum = register contents + all 0's + (carry_in = 1),

the boolean equations for a full adder can be simplified, since the B input to the adder is always zero. For each bit:

Sum = A xor B xor C = A xor 0 xor C = A xor C

Carry = A(A xnor B) + C(A xor B) = AC

The incrementer is thus implemented with a series of half-adders, as shown in Figure 13. Since the carry signal is propagated through a series of transmission gates, the carry must be buffered each four bits. The design of the half-adder is shown in Figure 14.

Note that in the least significant bit of the incrementer, the carry in will always be '1', so the equations for this bit simplify to:

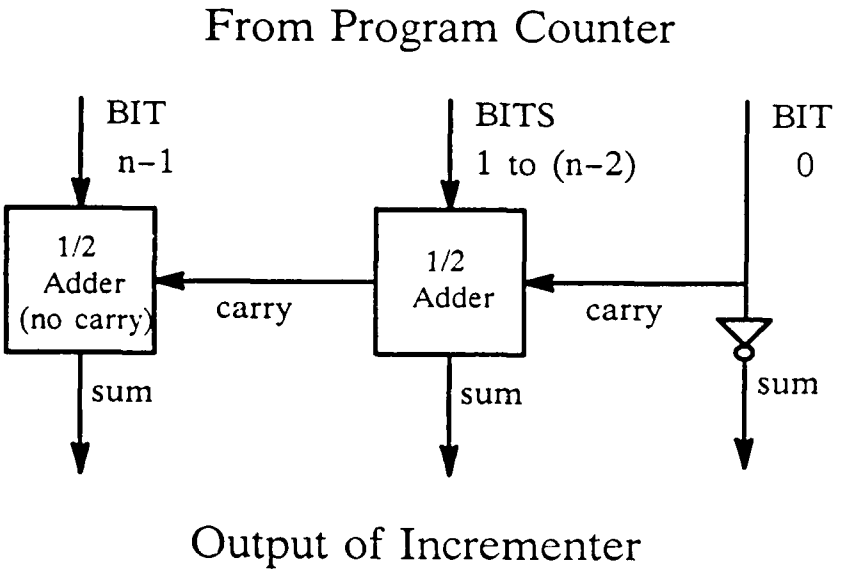


Figure 13. Incrementer

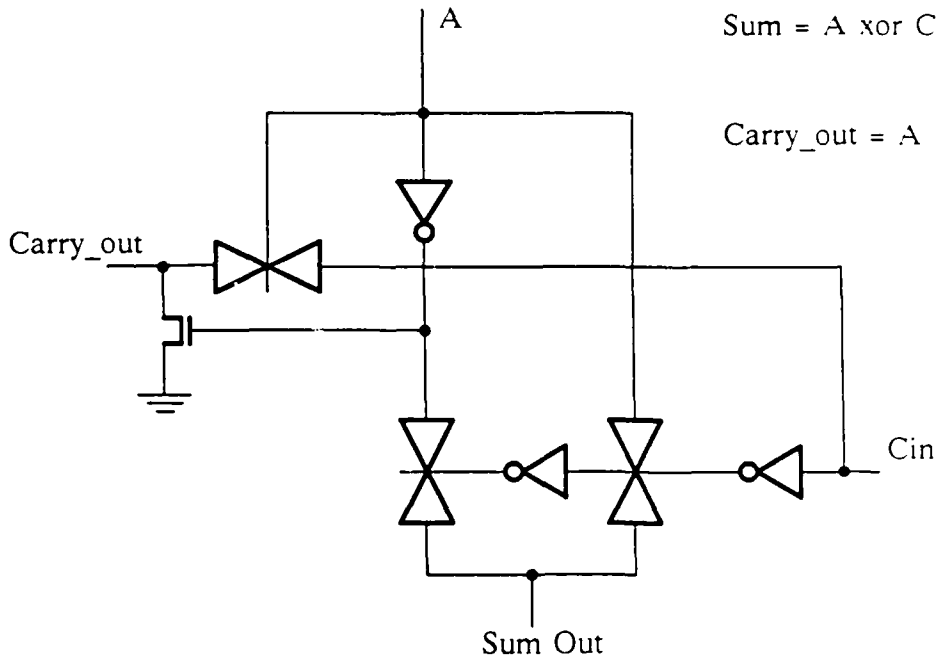


Figure 14. Half Adder

Sum = A xor C = A xor 1 = Abar

Carry = AC = A

Thus, only a single inverter is required to implement the LSB of the incrementer.

4.2.3.3 Address Multiplexer. As seen earlier, the input to the program counter can come from one of four sources. Since the design of the program counter itself allows the increment option to be loaded directly into the program counter, a means is required for selecting which of the other three options should be made available to the program counter. This is accomplished through the 4:1 address multiplexer. Although only three of the inputs of the 4:1 mux are actually required, the fourth input is provided to allow for future expansion of the microcode sequencer and to maintain

compatibility with the multiplexers in the condition multiplexer. Figure 15 shows the 4:1 multiplexer as designed by Lt. French.

4.2.3.4 Subroutine Stack. The subroutine stack is essential to the implementation of subroutine calls and returns. During subroutine calls, the contents of the program counter plus one (i.e. the output of the incrementer) must be "pushed" onto the last-in first-out (LIFO) stack. Conversely, when a return from subroutine is required, the top of the subroutine stack should be "popped" and loaded into the program counter. The width of the subroutine stack is determined by the width of the program counter. The depth of the stack will vary between various ASP implementations. The depth is determined by evaluating the target algorithm and determining the number of levels of nested subroutines required.

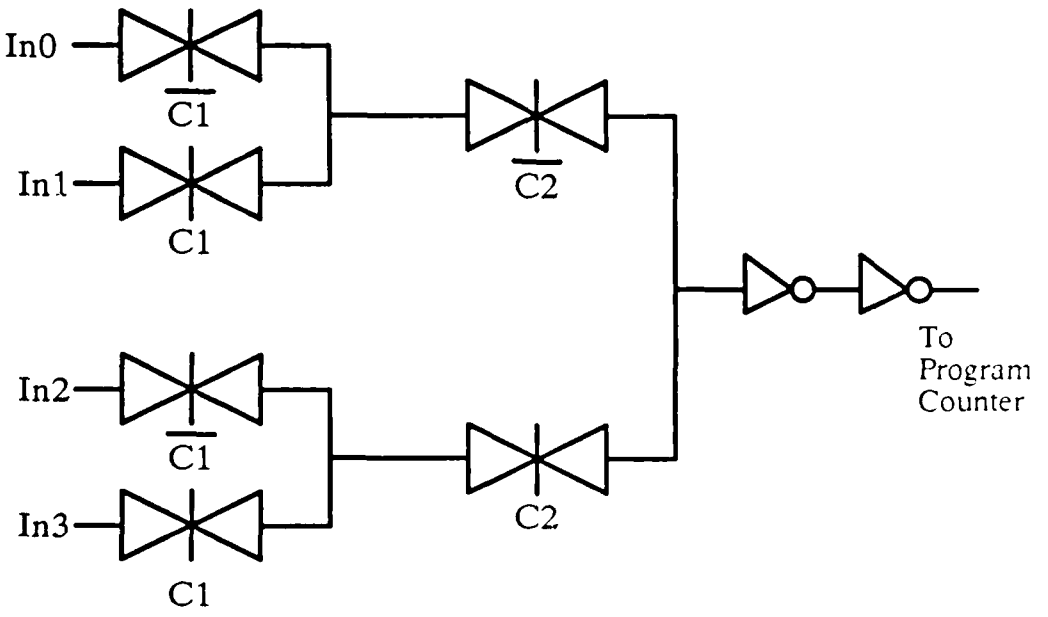


Figure 15. 4:1 Multiplexer

Figure 16 shows the design of the address stack. The "push" signal allows data to transfer between the MSFFs, moving deeper into the stack. Correspondingly, the "pop" causes the data to move toward the address bus. Twelve columns of MSFFs provide storage of the addresses. Although not required by most implementations, an extra column can be added to the stack to keep track of the number of words currently stored in the stack. This column is initially reset to all 0's during chip reset. If a "push" occurs, a '1' is pushed into the first MSFF in this column. Thus, a '1' stored in this MSFF serves as a flag that the stack is not empty. A '1' in the deepest MSFF in this column indicates that the stack is full. When the stack is popped, a '0' is shifted into the deepest MSFF.

The MSFF required for the stack (Figure 17) is similar to that required for the program counter. It must be loadable from two t-gates and be resettable (required for tag column only). The row of MSFFs which form the "top" of the stack require staged-up buffers to drive the address multiplexer when a "pop" command is received.

4.2.3.5 Branch Condition Multiplexer. The branch condition multiplexer is used to provide conditional branching within the microcode. The branch multiplexer is designed as a 32:1 mux. A 5-bit field from the ROM can then select one of 32 conditions as the branch condition. Flags from the datapath such as zero, overflow, or negative, or error conditions such as the IEEE not-a-number (NaN) can be used as branch conditions. Additionally, these conditions can be combined together with boolean logic to form such branch conditions as "greater than" or "less than or equal to".

The design of the 32:1 multiplexer (Figure 18) is simply a combination of 4:1 multiplexers, similar to the 4:1 multiplexers used in the address multiplexer. To lessen the capacitive loading on the 5 control lines, these lines are distributed over different stages

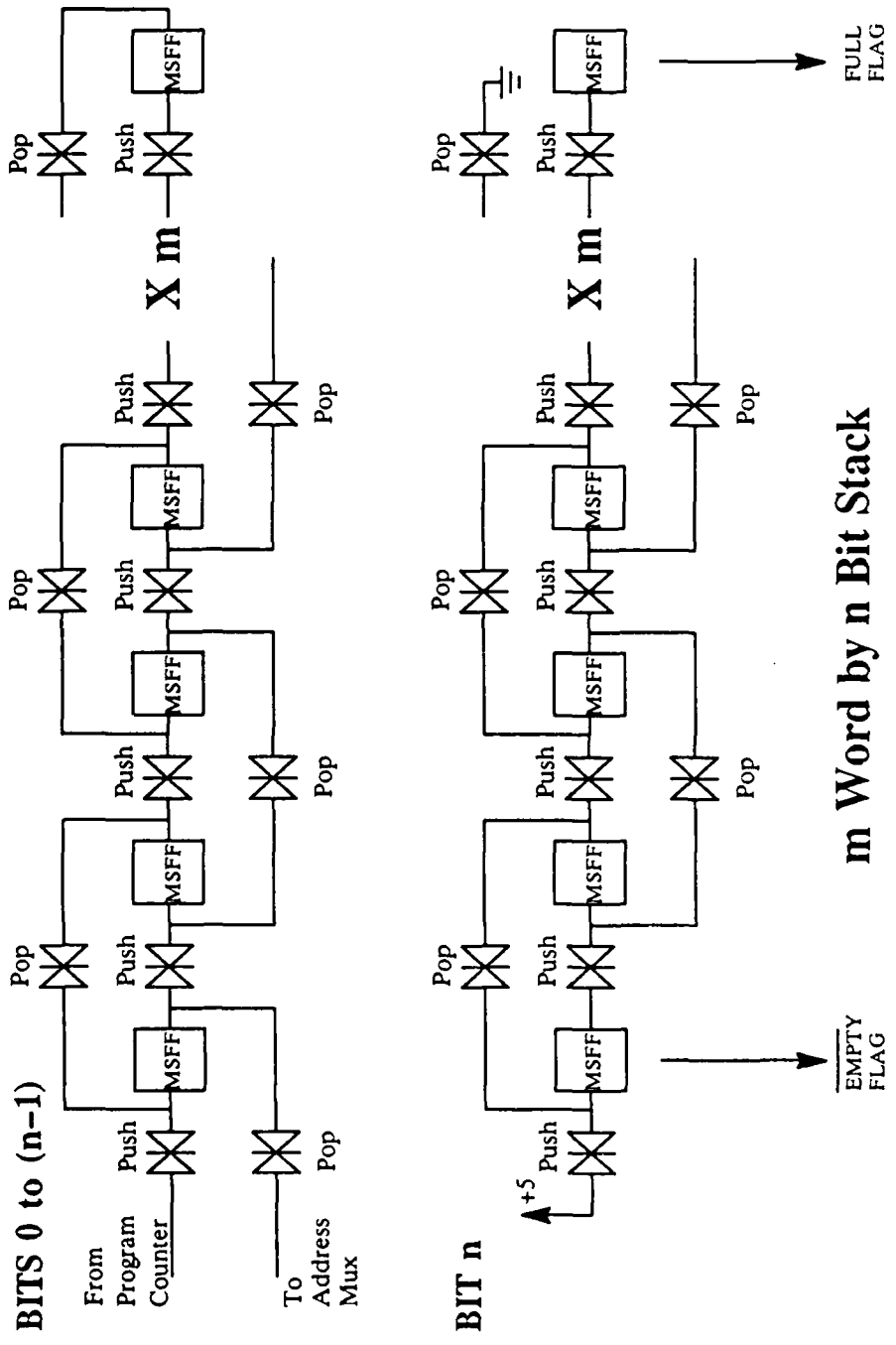


Figure 16. Address Stack

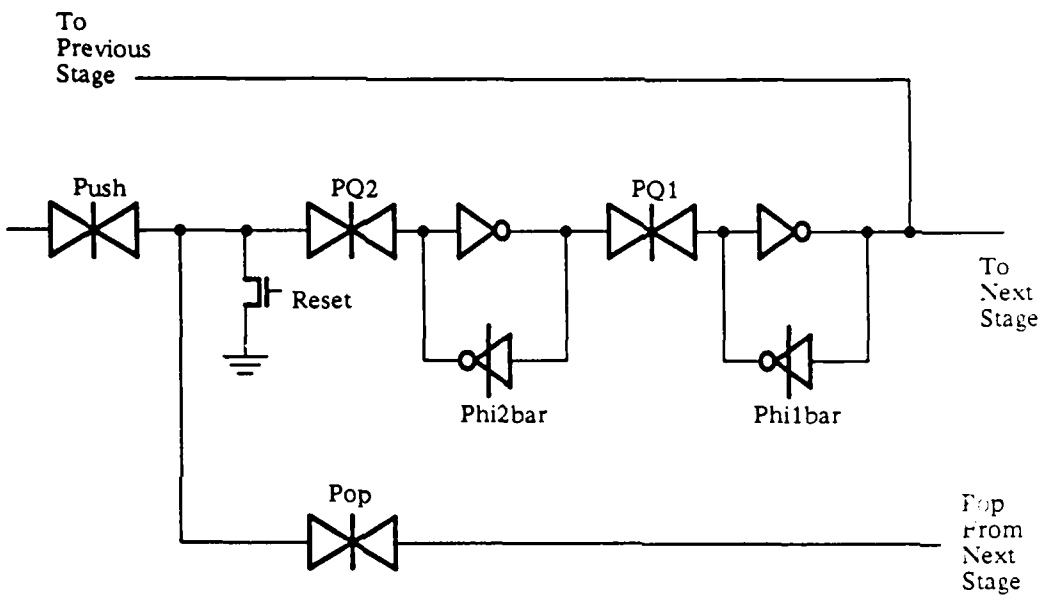


Figure 17. Stack MSFF

of the multiplexer, as described in [Fre86]. The effect is to decrease the delay through the multiplexer.

The polarity (active high/low) of the output of the 32:1 multiplexer is determined by a single bit coming from the XROM. The mux output and this bit, called the "BranchOn" bit, are XORed together to form the actual branch condition signal (Figure 19). A "high" on the BranchOn line, then, signifies a negative polarity on the branch condition. As an example, to form the condition "jump not zero", the 5-bit condition field would select the zero condition and the polarity bit would be set high, indicating a "not zero" condition. With the capability of branching on either polarity of 32 conditions, the branch condition multiplexer in effect provides 64 different branch options.

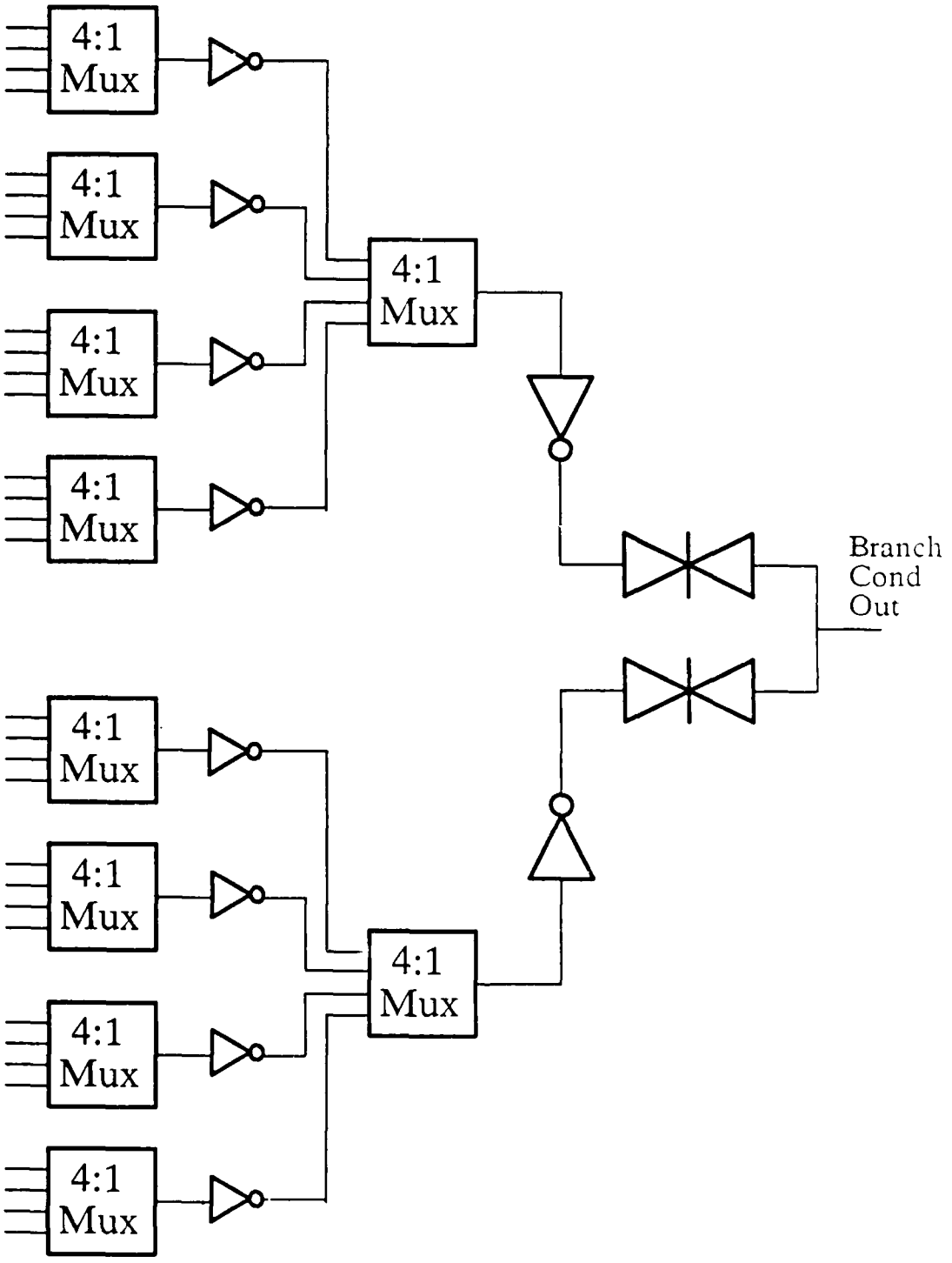


Figure 18. Condition Multiplexer

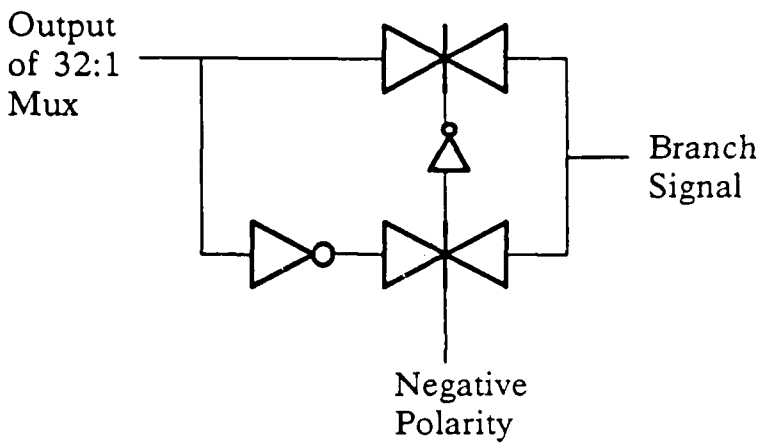


Figure 19. Branch Condition Signal

4.2.3.6 Sequencer Control. The subcells of the Microcode Sequencer described above require several control signals as input. The stack requires push pop signals. The program counter and address multiplexer require signals to determine which of the four next address options to load into the program counter. These signals are derived from a 3-bit field from the XROM, called the Next Address Field (NAF).

Table 2 shows the options provided by the Sequencer which can be selected by the NAF field. From this table, the Karnaugh maps can be developed for the required control signals and the boolean logic derived (Table 3). Implementation of the boolean logic to realize these signals is then trivial.

4.3 ASP Datapath Section

Whereas the function of the Control Section is to generate the required control signals for the ASP architecture, the function of the Datapath Section is to operate on the data in response to the signals. The datapath serves to transfer data to from off-chip,

Table 2. Microcode Sequencer Operations

NAF	Meaning	AddMux	MuxControl	Push	Pop
000	Continue	don't care	xx	0	0
001	Return	Pop Stack	10	0	1
010	Call	Branch Field	01	1	0
011	Branch	Branch Field	01	0	0
100	Cond. Datapath Load	Datapath	00	0	0
101	Cond. Return	Pop Stack	10	0	1
110	Cond. Call	Branch Field	01	1	0
111	Cond. Branch	Branch Field	01	0	0

Table 3. Microcode Sequencer Control Signals

Control Signal	Boolean Equation
Addmux1	NAF1
Addmux2	NAF1b * NAF0
Push	NAF1 * NAF0b * (NAF2b + BranchCond)
Pop	NAF1b * NAF0 * (NAF2b + BranchCond)
LoadPC	NAF2 * BranchCondbar + NAF2b * (NAF0 + NAF1)
IncrPC	LoadPCbar

store the data, and perform computations upon the data. Regardless of the type of data,

whether floating point, two's complement, signed magnitude, or some other format, the datapath is responsible for performing the required operations.

Figure 20 shows a block diagram of the ASP datapath. As depicted in this figure, each of the macrocells of the ASP datapath is designed such that the datapath can be laid out in a linear manner. The following sections examine the design of each of these macrocells in detail.

4.3.1 Datapath Busses. The ASP datapath is designed in a linear fashion, so that the data busses pass through the individual macrocells at a fixed spacing. This style, proposed by Mead and Conway [Mea81], allows the datapath to be a regular structure, as shown is Figure 21. For the ASP architecture, the intra-bus spacing is 81 lambda (lambda is a scalable size parameter used in VLSI design; a common size is lambda=1.5 microns), sometimes referred to as 81-pitch. This means that the spacing between different bits of the same bus is kept at a fixed spacing a 81 lambda. All macrocells which make up the datapath are designed to conform to this 81-pitch.

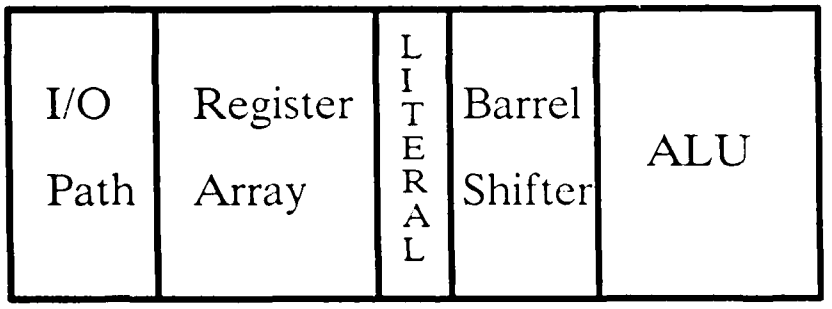
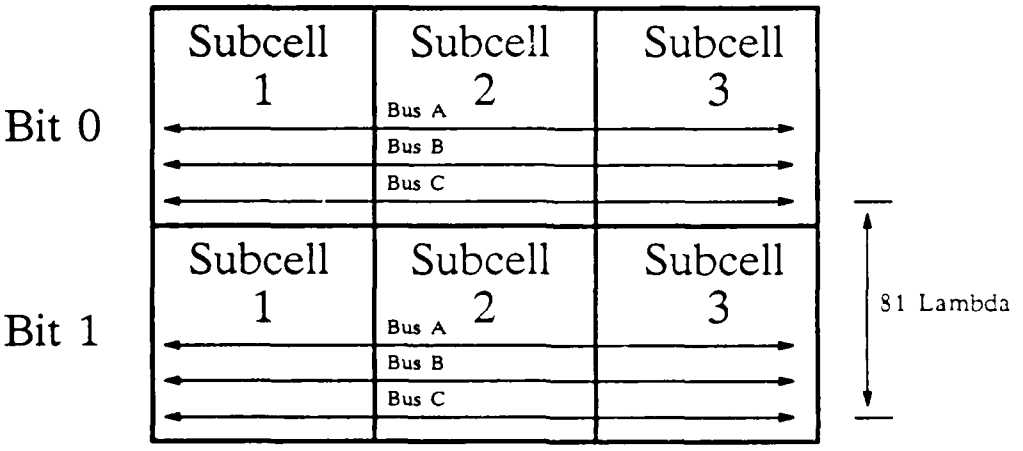
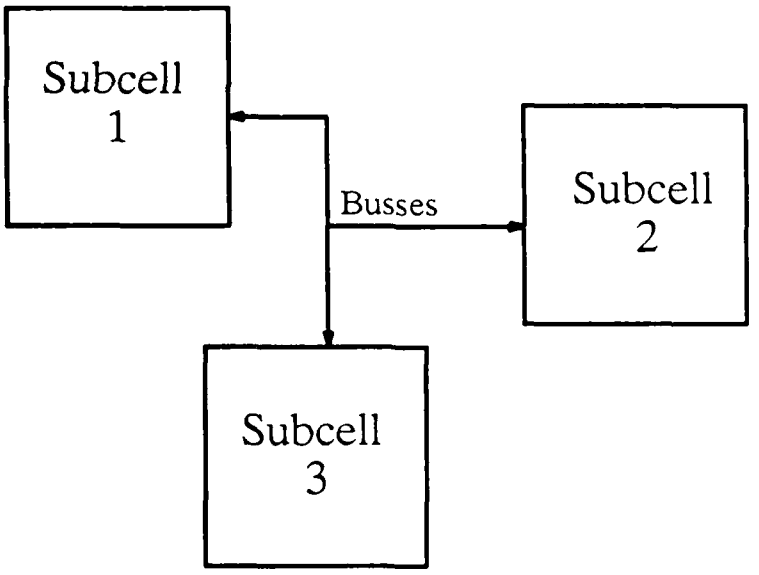


Figure 20. Block Diagram of ASP Datapath



a) Datapath Approach



b) Non-Datapath Approach

Figure 21. Datapath Pitch

The ASP architecture employs a three-bus structure (Figure 22). The "A Bus" and "B Bus" are used to transfer data from the storage devices (usually the register set) *to* arithmetic hardware (ALU and shifter), while the "C Bus" is responsible for returning results *from* the arithmetic hardware back to the registers.

Due to the high capacitive load on the datapath busses, various processor architectures apply different techniques in attempt to increase the speed of the busses. As seen in Chapter 2, precharging the bus and clamping the bus have been investigated. In CMOS design, another option is to provide large complementary drivers to circuits which must drive the bus. The ASP architecture utilizes a precharged design for source busses (Bus A and Bus B), and a conventional CMOS approach to the return bus (BUS C). Although precharging is less advantageous in a CMOS design than in an nMOS design (since CMOS employs p-channel pullups), the precharged approach resulted in smaller register cells and in less capacitive loading on the output driver control lines. The precharge signal must be timed such that it remains active until all register select control lines have settled to their new values, so that spurious select signals do not inadvertently discharge the busses. It was not practical to precharge the return bus, since the arithmetic hardware is not self-timed (it does not "signal" when the computation is complete).

4.3.2 Register Array. The general-purpose register array provides the primary data storage capability for the ASP datapath. As seen previously, the actual size of the register array for any given application is dependent upon the requirements of the algorithm. To simplify decoding, an array size which is some power of two, minus one, is preferred. The width of each register is set by the width of the datapath.

Figure 23 shows the design of the basic register cell. The heart of the cell is a MSFF, which drives new data on the rising edge of PQ1 and latches (loads) data on the

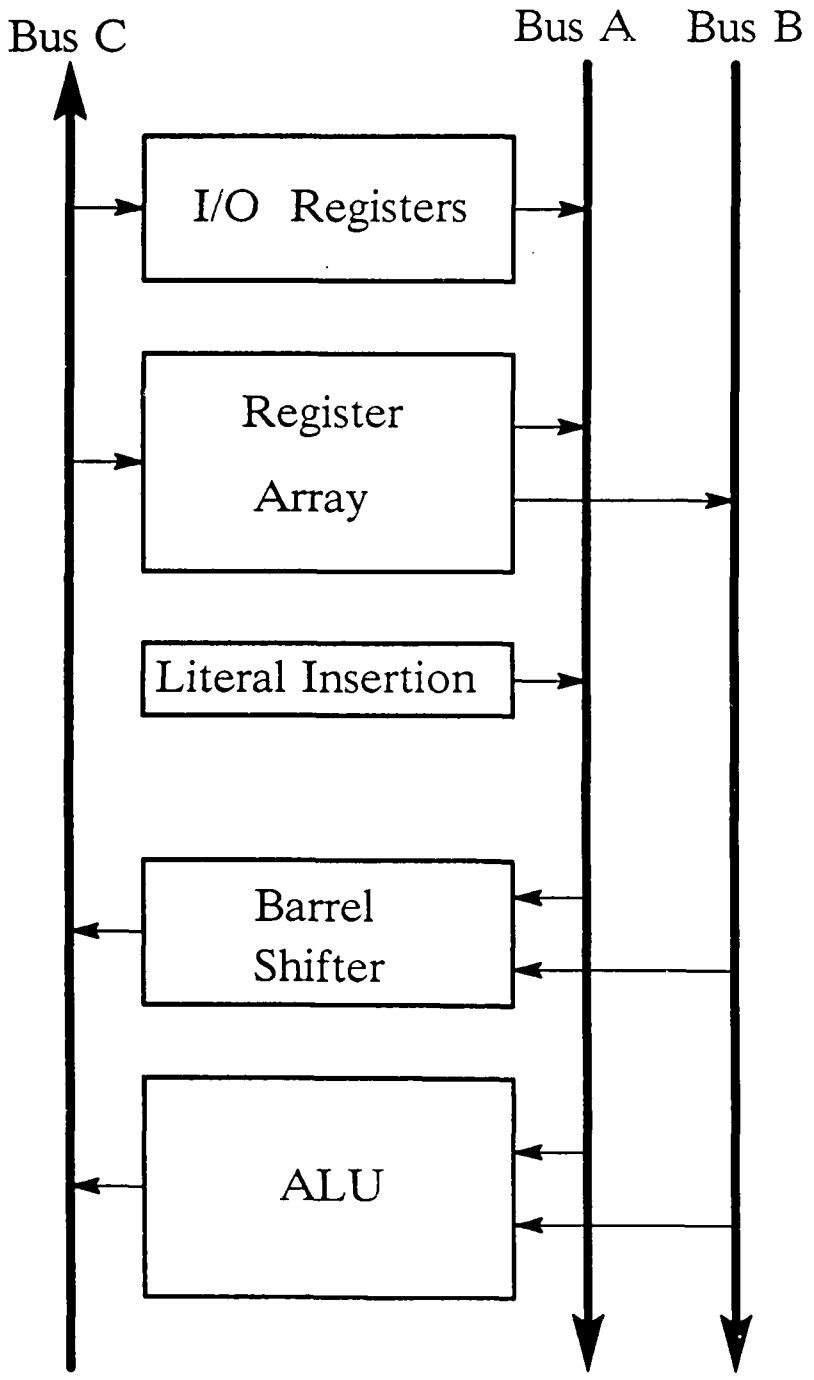


Figure 22. Datapath Bussing

trailing edge of PQ2 (Figure 24). The usable portion of each clock cycle, then, is from the rising edge of PQ1 (when new data is available to be driven onto the bus) until the falling edge of PQ2, when arithmetic results must be present at the input to the MSFF so that it will be latched. Actually, the usable period of time is reduced *slightly*, since the register drive signals cannot become active until the precharge signal is inactive.

The layout of the register cell diagramed in Figure 23 is set to the 81-pitch of the datapath. Since each register cell provides one bit, the register array is easily formed by laying out a regular array of cells, m bits high by n bits wide, where m is the number of bits in a word and n is the number of registers required.

The register array is controlled by three sets of decoders, one for each of the three busses. Two of the decoder sets select which one of the registers (if any) is enabled to drive the A Bus and the B Bus. The third decoder selects which of the registers will load *from the C Bus*. Regardless of its usage, the design of all decoders is the same. Obviously, the width of the decoders must vary with the number of general-purpose registers required. The selected register is decoded using a fully complementary NAND gate. In order to avoid unnecessary power dissipation, the output of the decoder must be ANDed

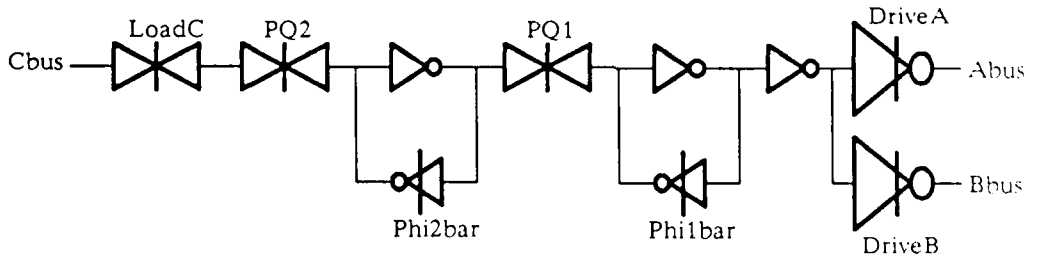


Figure 23. Basic Register Cell

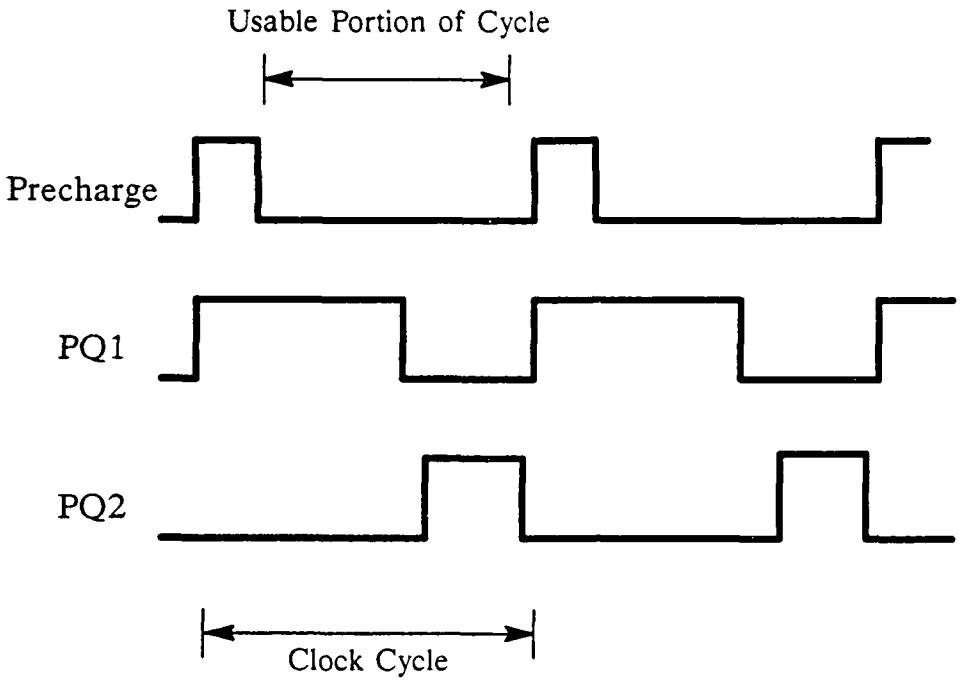


Figure 24. ASP Clock Cycle

with the inverse of the bus precharge signal (Figure 25). If the decoders were allowed to select a register while the busses were being precharged, the register pull-down driver would "fight" with the precharging circuit (if the register was attempting to output a '0'). The layout of the register decoder was patterned after the XROM address decoders, which allows for easy (and possibly automated) personalization of the decoder. Both polarities of each bit of the register select field are propagated through the decoders, so that personalization is accomplished by merely placing contacts at the appropriate intersection.

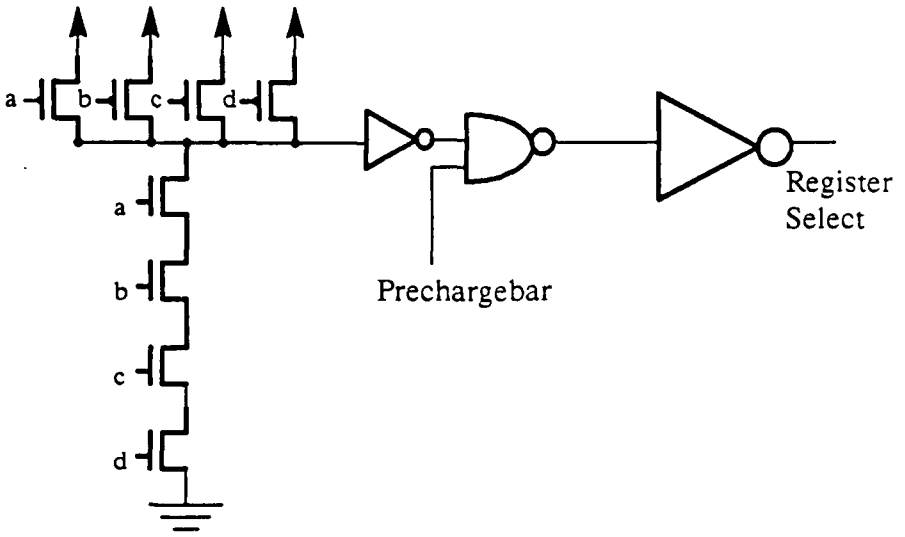


Figure 25. Register NAND Decoder

4.3.3 I/O Path. As described in Chapter 3, the I/O path consists of one or more I/O channels. Each of these channels supports both an address stream and a data stream. The address stream is provided by an address register, which directly drives output pads to control memory addressing. Figure 26 diagrams the design of the address register cell. This register is a simple modification of the general-purpose register cell. Like the standard register cell, the address register loads from the C Bus and can drive the A Bus. However, in place of a driver for the B Bus, the address register incorporates an ungated driver to the output pads.

The data stream is supported by the data register (Figure 27). Similar to the address register, the data register can load from the C Bus and drive the A Bus. Additionally, the data register can load from or drive bi-directional data pads. The given designs for the address and data registers do not provide the capability to drive data

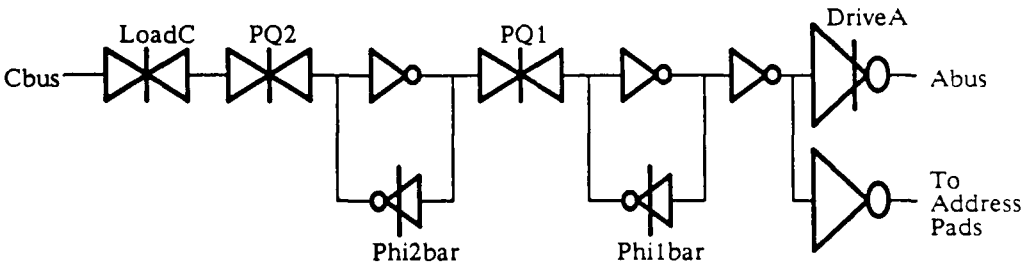


Figure 26. Address Register Cell

onto the B Bus. This capability could easily be added by employing another output driver, but was not implemented in order to minimize the size of these registers.

The address and data registers can each drive the A bus. Obviously, these registers cannot drive the A Bus at the same time one of the general-purpose registers is selected to drive the bus. The designer must therefore be careful in writing the microcode that this condition does not occur. The current microcode assembler does not provide an automated capability of deconflicting these instructions. In the case of loading from the C Bus, however, there is no contention. The I/O registers and general-purpose registers

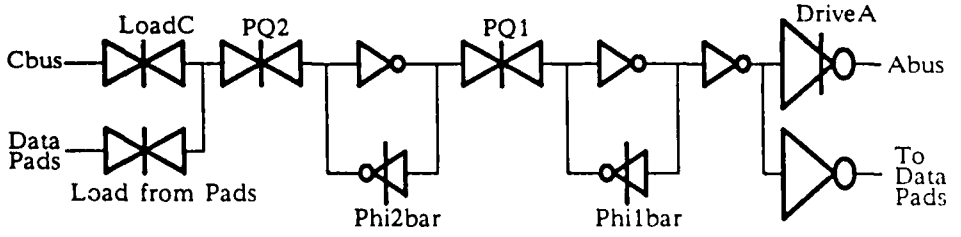
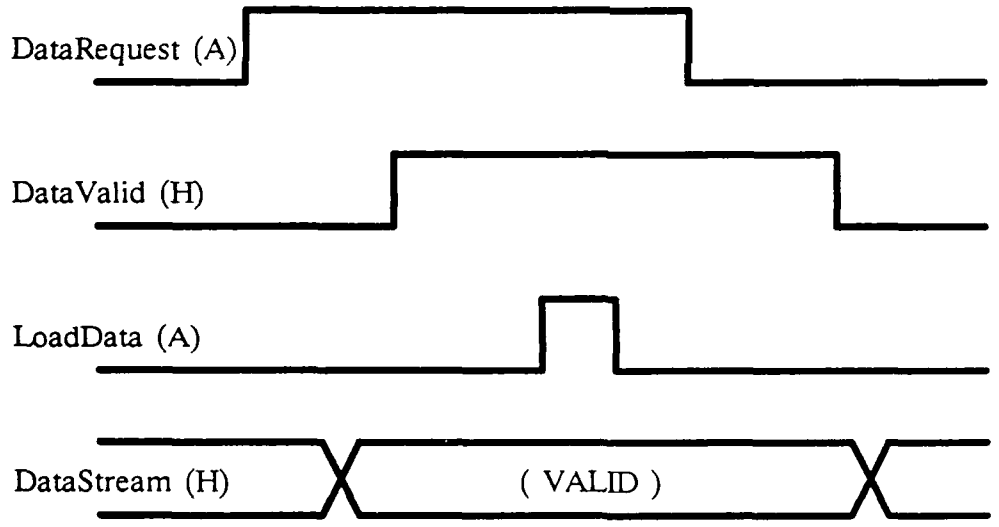


Figure 27. Data Register Cell

can load during the same clock cycle.

Two types of handshaking are envisioned for an I/O path: with a "host" controller or with memory. Data exchange between the ASP architecture and an intelligent host can be accomplished with a simple 4-line handshaking. After the ASP algorithm has been initiated by the host, the ASP hardware will initiate the handshaking for any necessary transfer. For data transfer to the ASP (Figure 28), the ASP will initially raise and maintain the DataRequest line. The host will then provide the necessary data on the data stream and raise the DataValid signal. Upon receipt of DataValid, the ASP loads the data register and then drops DataRequest. The host will maintain the data on the data stream until the DataRequest signal drops, at which time the host can release the data



(H) - Host Action (A) - ASP Action

Figure 28. Data Transfer to ASP from Host

stream and drop the DataValid line. For data transfer from the ASP to the host (Figure 29), the ASP drives the data register onto the data stream and then raises the DataAvailable signal. When the host has received the data, it sends DataReceived. The ASP can then lower DataAvailable and release the data stream.

Handshaking between the ASP architecture and a memory array is straightforward (Figure 30). A Read/Write line determines the direction of data flow along the data stream. A WriteEnable signal ensures that spurious data is not written while the address and data streams are unstable. This simple approach should place a minimal requirement on the memory hardware. If necessary, the designer can allow for a slow memory access time by inserting additional clock cycles into the microcode.

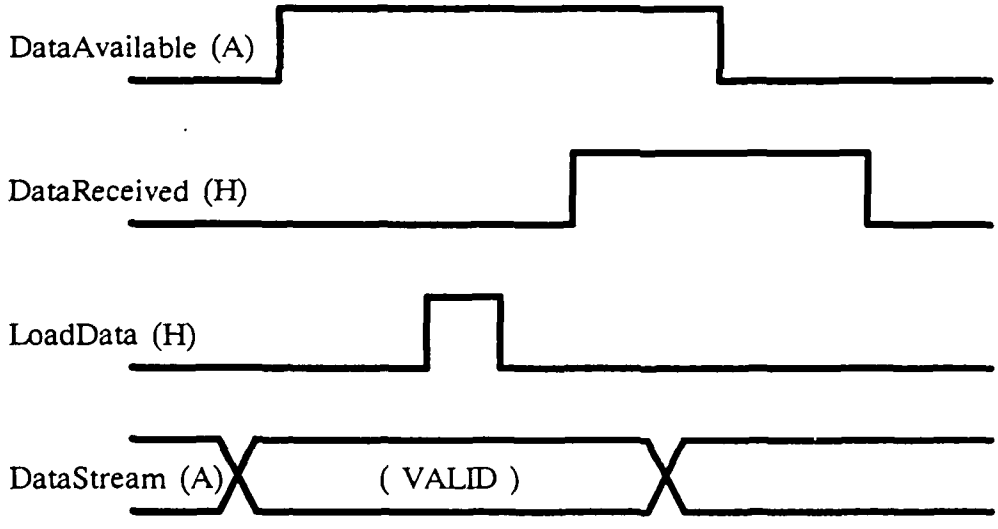


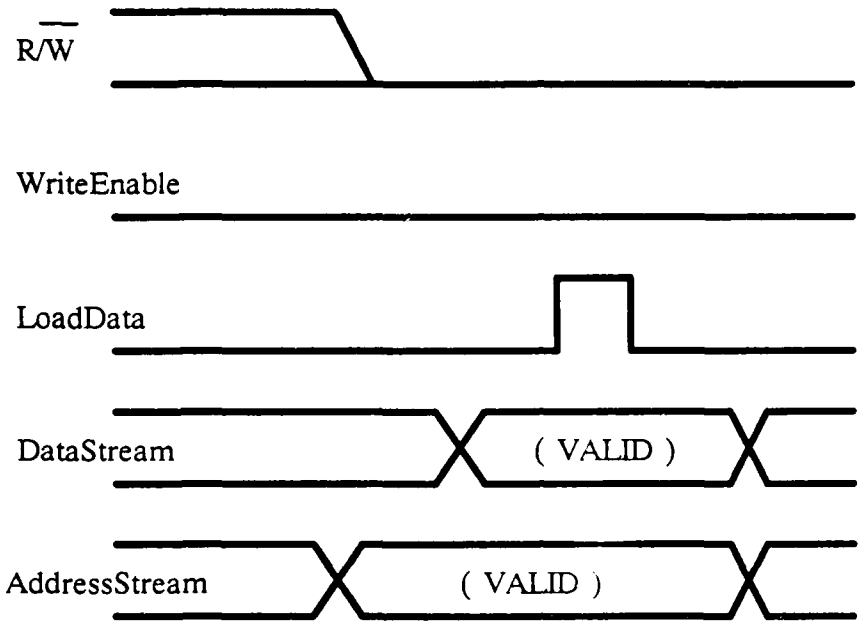
Figure 29. Data Transfer from ASP to Host

The ability to read data from memory allows the ASP architecture to load and execute assembly code instructions. This is accomplished in much the same manner as a CISC architecture. The instruction loaded into the ASP can be moved into an instruction register, from which the instruction is decoded into operation code (opcode) and register fields. The opcode portion of the assembly instruction is mapped into an address in the microcode ROM, using the Conditional Datapath Load microinstruction. Similar to a CISC, the assembly instruction is executed as a sequence of microoperations in the microcode store. Register mapping hardware decodes the register fields into the appropriate register select signals.

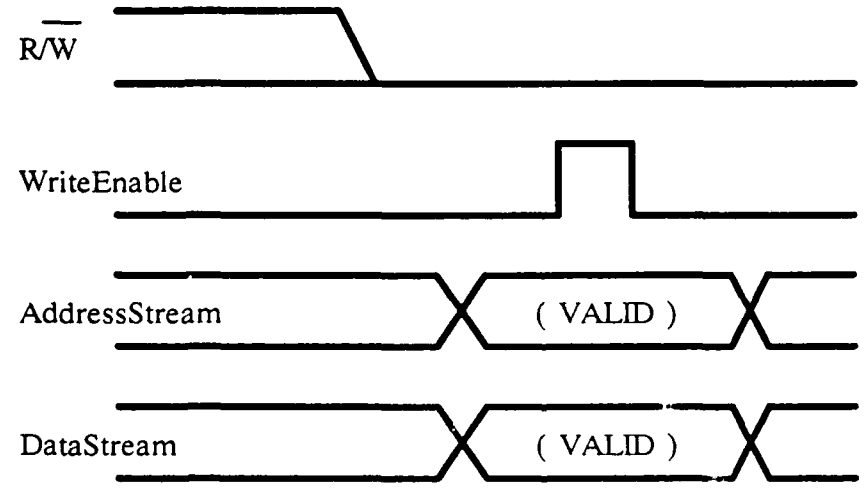
4.3.4 Literal Insertion. Although the I/O path allows the exchange of data between the ASP and a host, the majority of applications will use only a simple "Go-Done" handshaking. The ASP hardware must have the capability of providing the constants which are necessary for the algorithm. This is accomplished by storing constant data in a field of the ASP microcode XROM. Due to the width of the datapath, it is impractical to store an entire constant in a single XROM word.

Literal insertion is accomplished using the circuit in Figure 31. The literal field is inserted onto the LSBs of the A Bus if the InsertLiteral line is activated. If the designer wishes to pull the remaining bits of the A Bus low, the LiteralZero line is activated. Absence of the LiteralZero signal will cause the remainder of the bus to remain precharged high. The insertion of a literal onto the A Bus must be deconflicted with use of the A Bus by the register array and the I/O registers.

If a small constant, such as a loop variable, is all that is required, it can be inserted onto to A Bus and operated on during a single clock cycle. If insertion of a full-width constant is required, the constant can be loaded in two clock cycles. During the first



a) Read Cycle



b) Write Cycle

Figure 30. ASP Handshaking with Memory

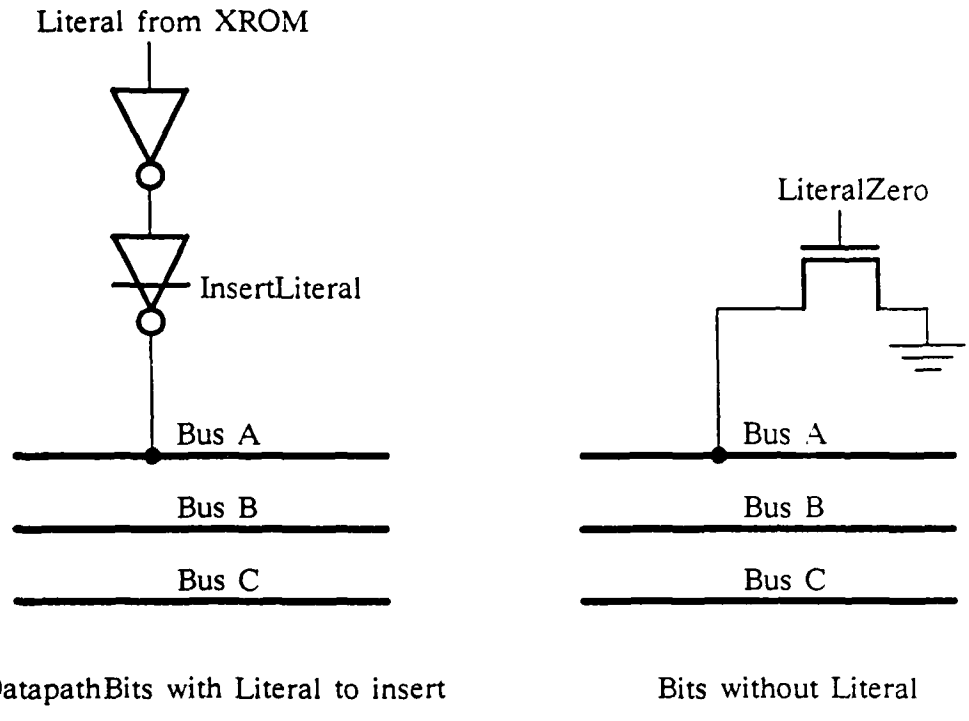


Figure 31. Literal Insertion Circuitry

clock cycle, the MSBs of the constant are placed on the A Bus and shifted into the most significant half of the datapath word. On the second clock cycle, the LSBs of the constant are inserted and ORed with the previously inserted bits. The completed constant is then ready for use during the subsequent clock cycle.

4.3.5 Barrel Shifter. Techniques for the design of barrel shifters are well established. The design of ASP barrel shifter is patterned after the shifters describes by Mead and Conway [Mea81]. Figure 32 shows the design of a simple 4-bit circular barrel shifter. The regularity of this structure facilitates the design of a standard shifter cell. This standard cell can then be arrayed in a n by n structure (with an 81-lambda pitch between bits) to provide an n-bit circular shift.

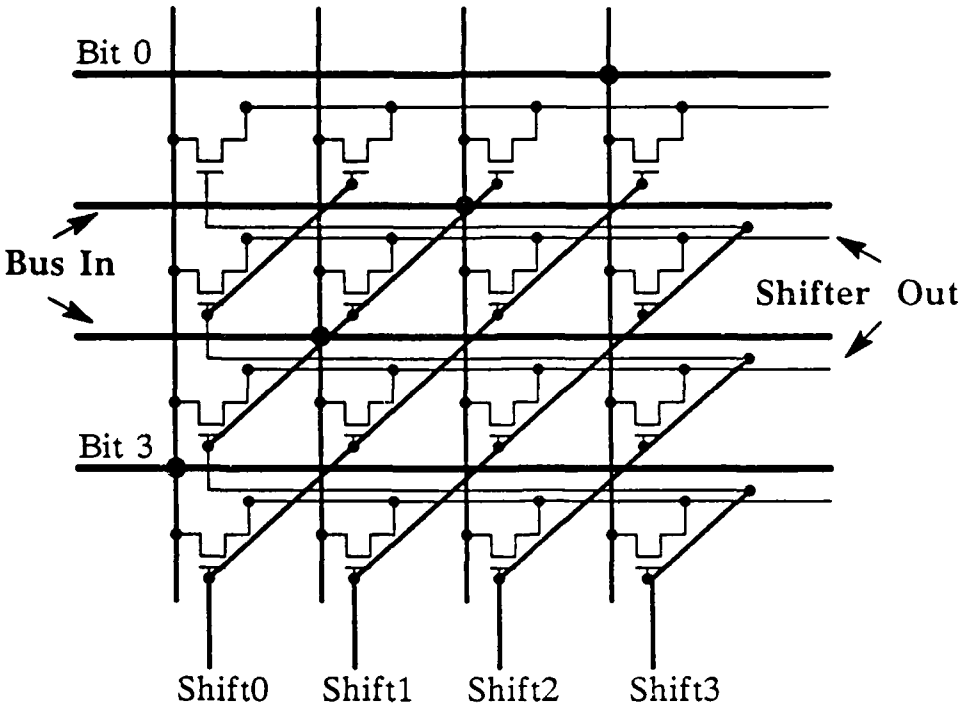


Figure 32. 4-Bit Barrel Shifter

If a circular shift is not required, only half of the shifter array is required. Figure 33 shows the 4-bit shifter, this time performing a left-shift, shifting in zeros into the least significant bits. This is the shifter design to provide an arithmetic left-shift. If the other half of the shift array is implemented, as in Figure 34, an arithmetic right-shift is implemented. In this case, the value of the MSB is shifted into the most significant bits. If a bi-directional arithmetic shift is required, the two halves of the shift array are both included in the ASP architecture. Due to their complementary shape, including both halves of the shift array will require little increase in area from the use of a uni-directional shifter.

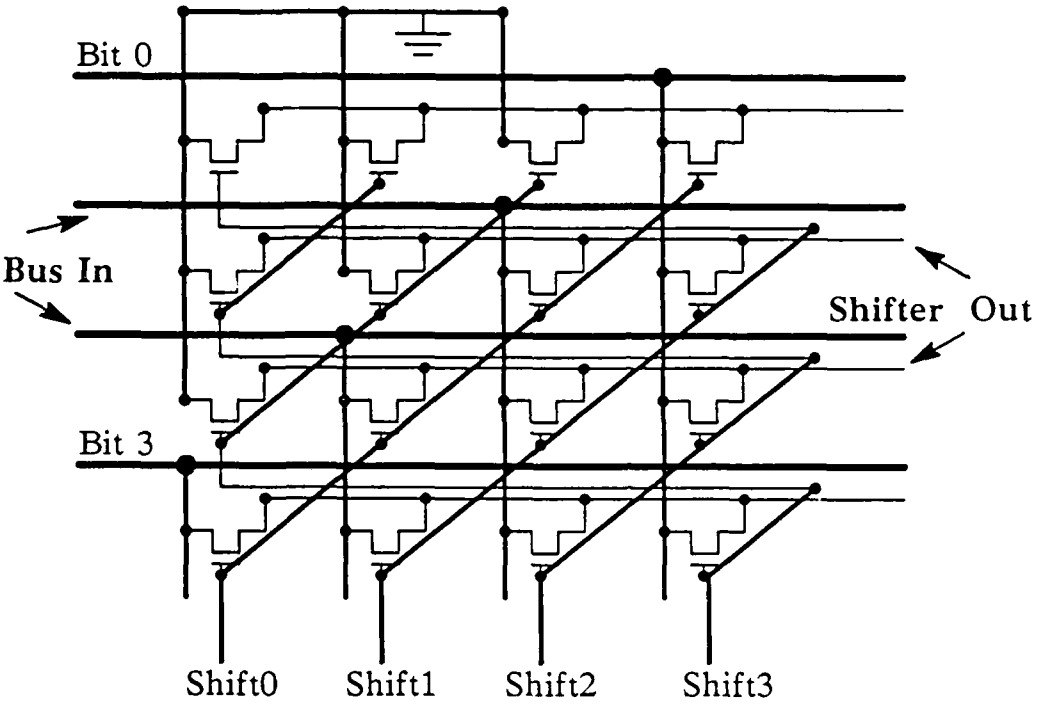


Figure 33. Barrel Shifter with Arithmetic Left Shift

Decoding for the barrel shifter is performed by a PLA. The PLA design used in the ASP architecture was initially developed for the WFT16 processor [Ros85]. The basic subcells for this design were obtained from the AFIT cell library. The PLA also provides a decoding for Shift0, which indicates that no shift is required. This signal is used by the shifter's bus interface logic (Figure 35) to determine whether the shift result is to be driven onto the result bus (C Bus). In order to deconflict with other arithmetic hardware, it is important that the shifter only drive the C Bus when a shift is selected.

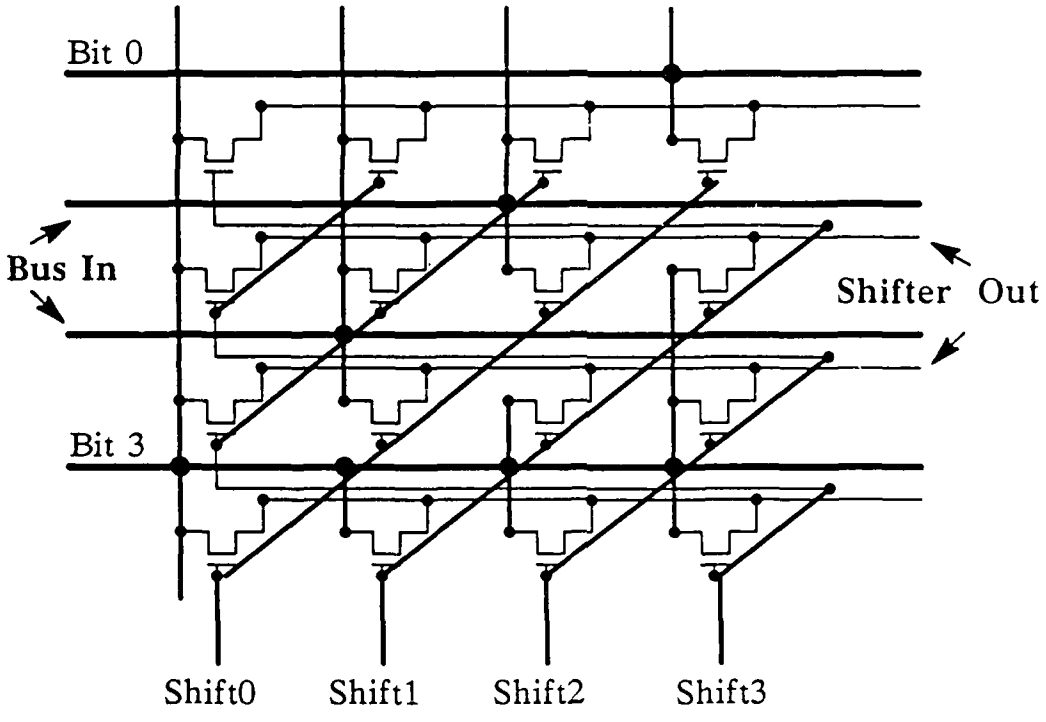


Figure 34. Barrel Shifter with Arithmetic Right Shift

4.3.6 Integer ALU. For applications that do not require floating-point arithmetic, the integer ALU is responsible for providing all arithmetic processing. ASP architectures which process primarily floating point data will still require an integer ALU for address computations and to support software operations such as looping. The integer ALU should provide the following functions:

1. Addition/Subtraction
2. Increment/Decrement
3. Boolean Logic - AND, OR, XOR, Complement
4. Set/Reset Carry
5. Move Between Registers

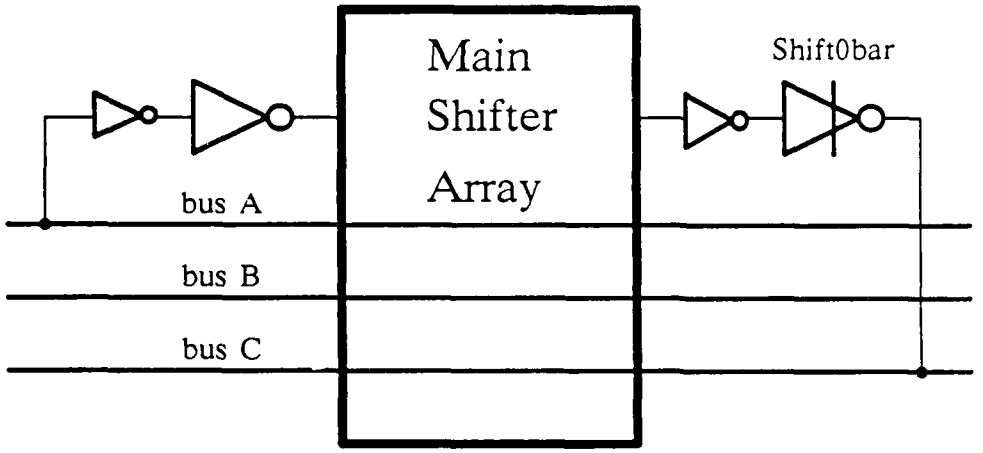


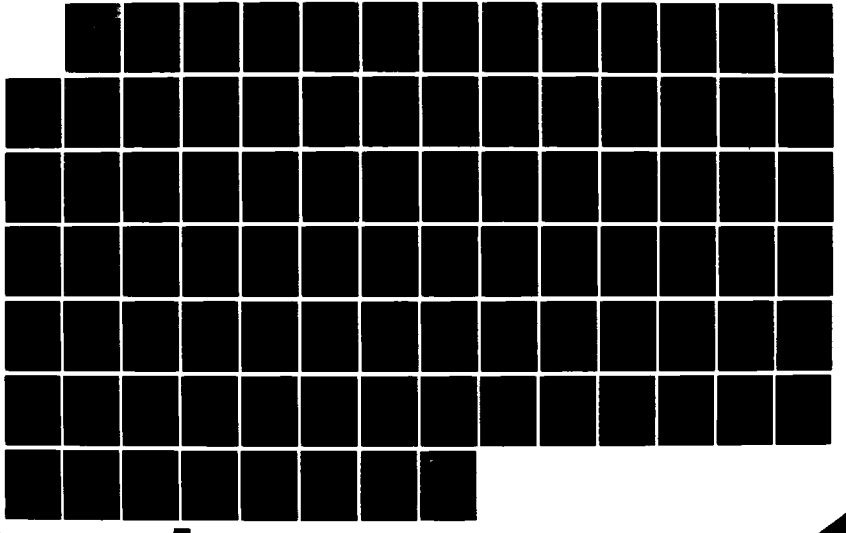
Figure 35. Shifter Interface to Busses

Figure 36 shows a block diagram of a design to provide the required functions. Processing is accomplished by five functional units, which perform the addition, AND, OR, XOR, and pass (move) functions. The A Bus feeds these five functions directly. The B Bus, however, is fed into the B select unit, which sends one of four possible values (B, B_bar, 0, 1) to the functional units, dependent upon the function desired. The third input to the ALU is provided by the carry in logic, which can input carry, carry_bar, 0, or 1 into the ALU's carry in. A 5:1 multiplexer selects which of the five functions will be routed onto the C Bus. Additionally, the ALU will generate signals for four flags: Zero, Negative, Overflow, and Carry.

The hardware to realize the boolean functions is trivial. Figure 37 shows the design of these units. Due to the simplicity of this boolean hardware, and the fact that the computations on one bit are independent of all other bits, none of these units is a critical timing path.

AD-A189 541 RAPID PROTOTYPING OF APPLICATION SPECIFIC PROCESSORS (U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING M GALLAGHER DEC 87 2/2

UNCLASSIFIED AFIT/GE/ENG/87D-19 F/G 12/6 NL



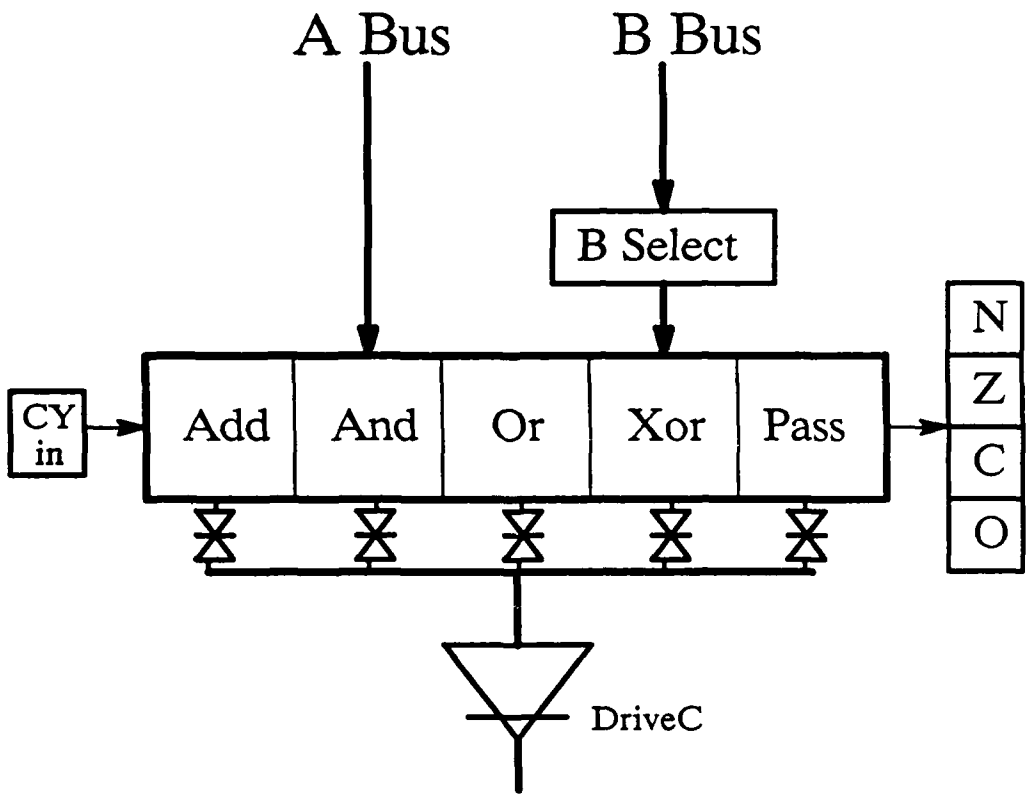


Figure 36. ALU Block Diagram

4.3.6.1 Integer Adder. The adder circuitry, therefore, establishes the critical timing path through the integer ALU. In fact, since the barrel shifter is very fast and any floating point operations will probably be accomplished over 2 clock cycles, the delay through the integer adder becomes the critical path through the entire arithmetic section of the datapath. The delay through the adder must be minimized.

For a simple ripple-carry type adder, addition of the input bits does not become effective until carry signal from the previous bit becomes valid. To speed up the carry signal, carry propagate adders compute the carry-out signal rapidly, so that the carry

signal is propagated faster than the sums are computed. The carry out of a block of several bits can be computed even faster using carry lookahead adders [Wes85]. In this scheme, the words to be added are broken up into blocks of bits. The carry-out of a block of bits is computed entirely separately from the sums, using a static boolean logic design.

An even faster adder can be built using the carry-select adder approach, which significantly increases the area required by the adder. The carry-select approach relies upon the fact that all the bits to be added become valid at the inputs at approximately the same time. The carry-select adder can thus begin adding all bits at the same time. Using this approach, the words to be added are again broken up into blocks, as shown in

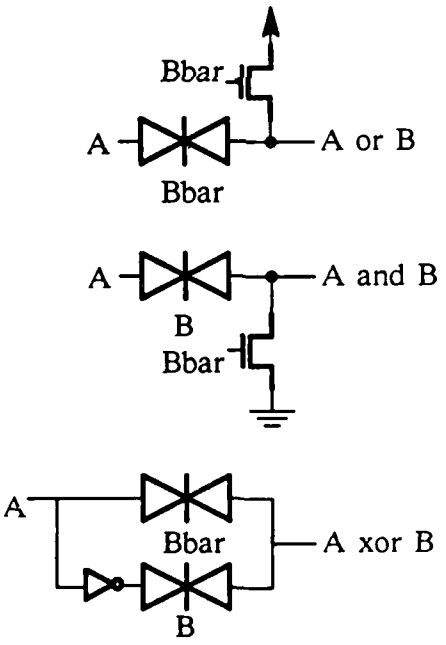


Figure 37. Implementation of Logic Functions

Figure 38. For each block, or stage, of the adder after the least significant block, two adders are provided for each bit. One of the adders computes its sums and carry-out based on a carry-in of zero, while the other computes based on a carry in of one. Once the previous stage computes its carry-out, this stage merely has to select one of its two possible carry-out signals via a 2:1 multiplexer. Thus, the delay from carry-in to carry-out of the block is merely the delay through the mux: a single one logic delay. In a same manner, the proper sum value can be selected by the carry-in of the stage.

The ASP integer ALU employs the carry-select adder approach. This approach results in a fairly constant add time for various widths of datapath, since adding an additional stage of 4-6 bits requires only one additional logic delay to compute. The adders within each block are designed using a carry propagate style. Figure 39 shows the design of this adder. The adder is built using a t-gate design and introduces only a single t-gate delay from carry-in to carry-out. The adder is designed using a slightly different form of the full-adder equations than is normal. The most common equations for the sum and carry of a full adder are:

$$\text{sum} = A \text{ xor } B \text{ xor } C$$

$$\text{carry} = AB + AC + BC$$

However, analysis of the Karnaugh map for a full adder reveals an alternate form for the carry equation:

$$\text{carry} = (A \text{ xor } B)C + (A \text{ xnor } B)A$$

From this form of the equation, it is seen that the carry signal can be generated by a simple multiplexer, based on the results of the XOR and XNOR operations. The adder in Figure 39 uses XOR gates to compute "A xor B" and "A xnor B", then uses these sig-

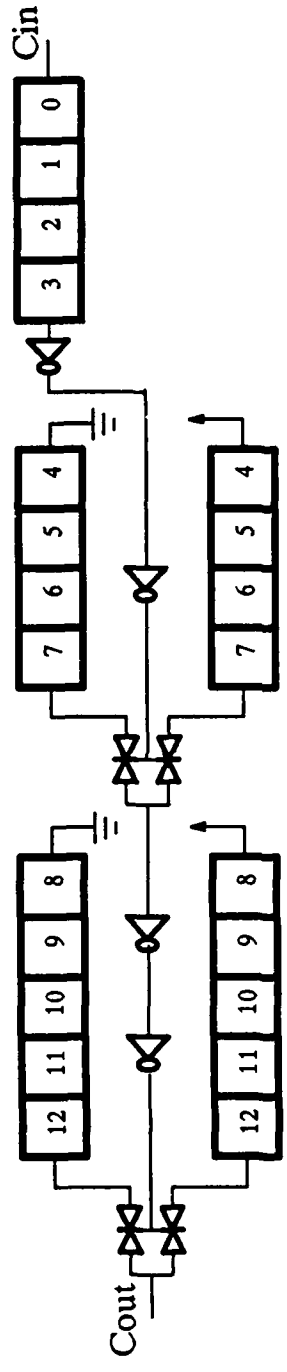


Figure 38. Carry-Select Adder

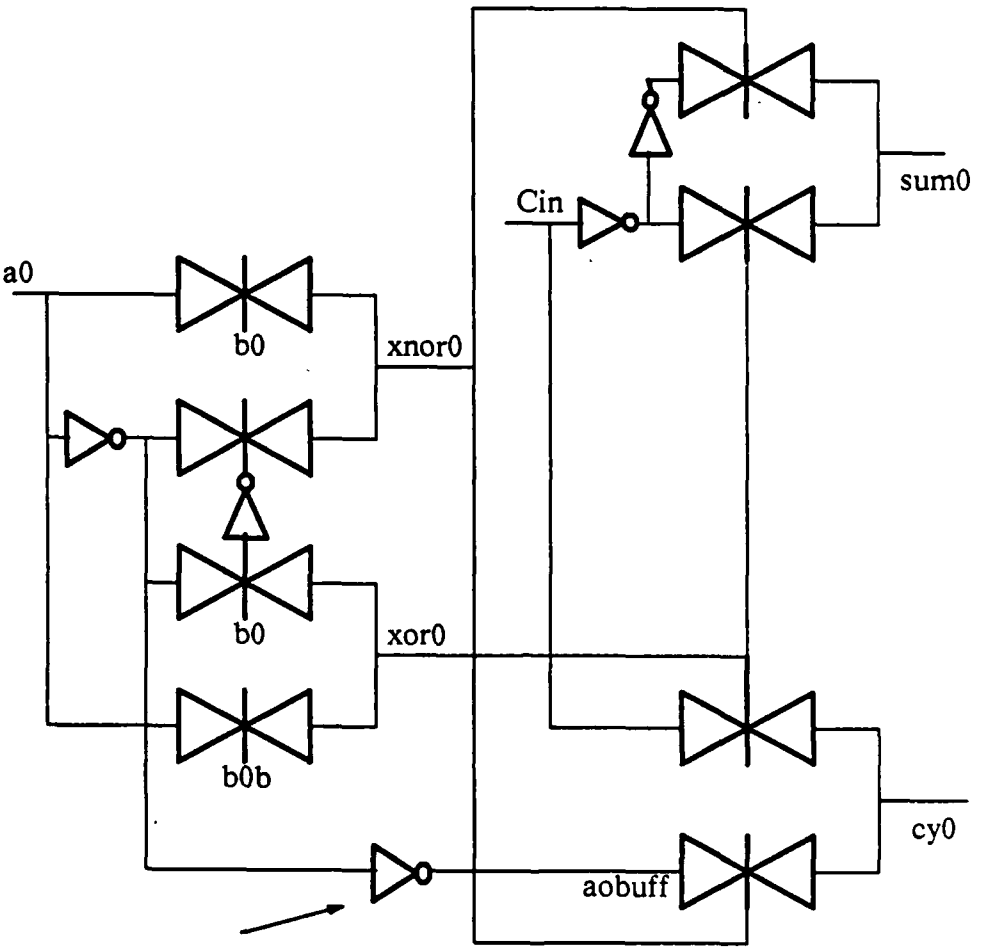


Figure 39. Carry Propagate Adder Cell

nals to select the sum and carry. Since the XOR and XNOR signal can be computed prior to the arrival of the carry-in signal, carry delay through the adder is only one t-gate delay, which is comparable to the delay of a carry lookahead adder. Note in Figure 39 that the A signal into the carry mux in generated by inverting A_bar, rather than using the A signal already present. This extra inverter was added so that the ESIM switch simulator would not be confused by what is construes as feedback in the circuit.

Using this carry-select adder approach, the ASP architecture can perform a 32-bit add in approximately 24 ns (worst case, assuming a 3 micron CMOS fabrication process), from stable inputs until stable outputs of the adder. Overall add time for the integer ALU will be longer, due to overhead required to select the B input (prior to the add) and to drive the result through the 5:1 mux and buffer it onto the C Bus (following the add).

4.3.6.2 ALU Functions. Table 4 shows a listing of how desired operations are realized from the ALU hardware. Simple addition is accomplished using a carry-in of 0, while the add-carry operation uses a carry-in based upon the status of the carry flag. Subtraction is realized by complementing B and adding A, with a carry-in of 1. Subtract-with-borrow is not immediately obvious. If borrow is set (actually, the carry flag), then the difference would be A + B_bar; but if borrow is not set, the difference is A + B_bar + 1. Thus, the proper equation for subtract-with-borrow is A + B_bar + Carry_bar. Then increment/decrement functions and most of the boolean functions are fairly obvious. The complement function is realized by using the XOR function. The set and reset carry functions were accomplished by a direct set/reset of the carry flag register. An alternate solution would have been to use the following equations:

set carry = A + (all 1's) + 1 (always resulting in carry-out)

reset carry = A + (all 0's) + 0 (never resulting in a carry)

The move function could also have been provided by:

move = A or (all 0's)

Table 4. ALU Operations

Operation	Implementation
Invert A	C = A xor (all 1s)
AND	C = A and B
XOR	C = A xor B
OR	C = A or B
Move	C = A
Set Carry	Directly Set Flag Reg.
Reset Carry	Directly Reset Flag Reg.
Incr	C = A + (all 0s) + 1
Decr	C = A + (all 1s) + 0
Adc	C = A + B + cy
Add	C = A + B
Sub	C = A + Bbar + 1
Sbb	C = A + Bbar + cybar
Compare	A + Bbar + 1

4.3.6.3 ALU Control Circuitry. Table 5 shows the integer ALU functions, showing the necessary operation, B select, and carry-in to realize the function. The table also shows which flags will be modified by each of the functions. The functions have been reordered to simplify the boolean equations for control signals. From Table 5, Karnaugh maps can be drawn for the required ALU control signals. Table 6 shows the boolean equations which were derived from the Karnaugh maps. Simple implementation

of this boolean logic provides the necessary ALU control signals.

Table 5. Control Requirements of ALU Operations

ALU 3-0	Operation	BSelect	Function	CyIn	Flags
0000	NOP	X	X	X	-
0001	C = Abar	1	XOR	X	Z
0010	C = A and B	B	AND	X	Z
0011	C = A xor B	B	XOR	X	Z
0100	C = A or B	B	OR	X	Z
0101	C = A (mov)	X	Pass	X	-
0110	Set Carry	X	X	X	C
0111	Reset Carry	X	X	X	C
1000	C = Incr A	0	ADD	1	All
1001	C = Decr A	1	ADD	0	All
1010	C = A + B + cy	B	ADD	Cy	All
1011	C = A + B	B	ADD	0	All
1100	Not Defined	X	X	X	X
1101	C = A - B	Bbar	ADD	1	All
1110	C = A - B - borrow	Bbar	ADD	Cybar	All
1111	Compare A, B	Bbar	ADD	1	All

Table 6. ALU Operations

Function	Control Sig	Boolean Logic
Bselect	0	ALU2b * ALU1b * ALU0b
	1	ALU2b * ALU1b * ALU0
	B	ALU3b * ALU2 + ALU2b * ALU1b
	Bbar	ALU3 * ALU2
Function	Pass	ALU3b * ALU2b * ALU0
	XOR	ALU3b * ALU2b * ALU0
	OR	ALU3b * ALU1b * ALU0b
	AND	ALU3b * ALU1 * ALU0b
	ADD	ALU3
Carry In	Cy	ALU2b * ALU1 * ALU0b
	Cybar	ALU2 * ALU1 * ALU0b
	0	ALU2b * ALU0
	1	ALU2 * ALU0 + ALU2 * ALU1b + ALU1 * ALU0b
DriveC	Drive	ALU3 + ALU2 + ALU1 + ALU0
LoadNegFlag	Load	ALU3
LoadOverFlag	Load	ALU3
LoadCarryFlag	Load	ALU3
LoadZeroFlag	Load	ALU3 + ALU2b * ALU1 + ALU2 * ALU1b * ALU0b

4.3.6.4 ALU Flags. The integer ALU flags are easily obtained. Each of the

flags is stored in a register cell, similar to the basic register cell designed for the register array. The carry flag register (Figure 40) additionally requires a set/reset capability. The Zero Flag is obtained using a multiple-input NOR gate which samples the output of each bit of the ALU. Since the ALU assumes two's complement arithmetic, the Negative Flag is the output of the ALU's MSB. Overflow is detected when the carry-out of the two MSBs of the ALU is not the same. Therefore, the Overflow Flag is obtained from an XOR of these two signals. The Carry Flag is derived from the carry-out of the MSB. For addition-type operations (add, adc, or incr), the MSB carry-out signifies ALU Carry. However, for subtraction operations (sub, sbb, compare, or decr), the MSB carry-out signifies Carry_bar. Therefore, the Carry Flag is derives as the XOR of the MSB carry-out and a signal indicating a subtract- type operation (ALU2 + ALU1bar * ALU0). The load conditions for these flags was shown in the previous section.

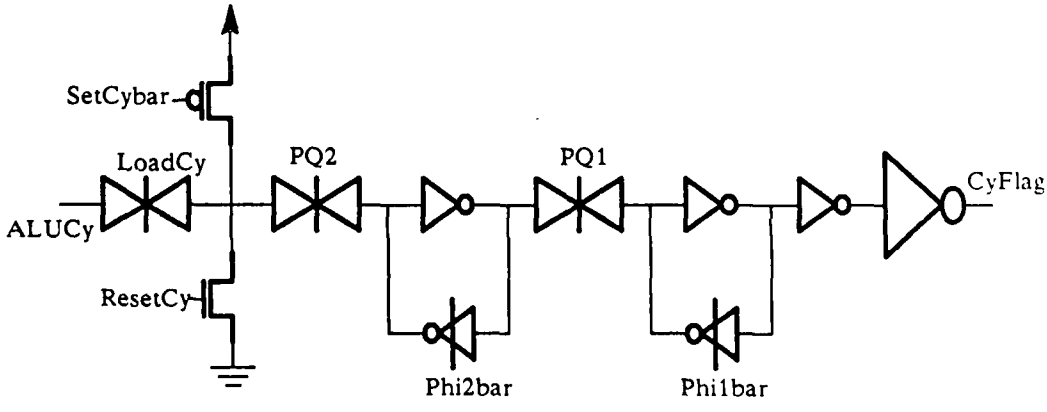


Figure 40. Carry Flag Register

4.3.6.5 ALU Integer Multiply. Many integer applications will require a multiply function from the ALU. Most processors today perform the integer multiply in software, using a series of shifts and adds. Some processors provide limited hardware support for the software multiply, in the form of a special "add and shift" instruction which employs Booth's modified algorithm to halve the number of adds required.

This approach is desirable if multiplies are required infrequently. However, if the application's algorithm requires frequent multiplies, further hardware support is warranted. A hardware multiplier can perform the multiply in less than 100 ns. A hardware multiplier has therefore been designed for the ASP architecture. This integer multiplier is incorporated into the floating point multiplier which is described in Appendix A. the multiplier will multiply two 24-bit integers, producing either a 24-bit or 48-bit product. The main multiplier array employs Booth's modified algorithm in a Wallace tree structure to provide an integer multiply time of less than 75 ns.

4.3.7 Floating Point Hardware. The ASP architecture's primary hardware support for floating point data is in the form of a floating point adder and multiplier. Both the adder and the multiplier perform floating point calculations based on the IEEE standard floating point format [IEE85]. The design of the floating point multiplier, which has been implemented and tested, is described in detail in Appendix A.

Design of the floating point adder was initiated as a EE695 class project during the Winter 1987 quarter [God87]. Design goals for the adder was a 50 ns add time, accomplished within a 2000 X 3000 lambda area. Since the ASP integer hardware is designed for a 50 ns clock cycle, it would be impractical to drive data to the adder, perform the floating point addition, and return the result to the register array within one clock cycle. The floating point addition will thus require two clock cycles. The data will be latched

into input registers during the first clock cycle and the result will be available during the subsequent clock cycle.

The architecture of the floating point adder logically divides into two sections: the exponent section and the mantissa section. Figure 41 shows the design of the exponent section. The first step in the exponent section is to subtract the two exponents to determine which is greater. The two's complement circuitry ensures that the result is in signed-magnitude form. This signed-magnitude result is passed to the mantissa to indicate which mantissa should be denormalized and how far to shift this mantissa. The larger of the two exponents is then gated to the renormalization adder. If renormalization is required after the mantissa addition, the exponent is added to the renormalization amount to determine the correct exponent to output.

The mantissa section (Figure 42) must await the exponent comparison before it can begin processing. This comparison tells the mantissa section which mantissa to gate into the barrel shifter and how far it must be shifted for denormalization. After the proper mantissa has been denormalized, a 25-bit full adder will either add or subtract the mantissas, dependent upon the two sign bits. A good description of the addition/subtraction algorithm for signed magnitude numbers can be found in [Man82]. The output of the adder is then converted back to signed-magnitude format and tested for overflow or leading zeros, either of which would require renormalization. If renormalization is required, the amount is passed to the exponent section and the mantissa is renormalized via a shifter. Contradictory to [God87], the shift will not always be left, since in the case of overflow the mantissa will have to be shifted one right.

The critical timing path through the floating point adder is through the exponent difference circuitry, then through mantissa denormalization, through mantissa addition,

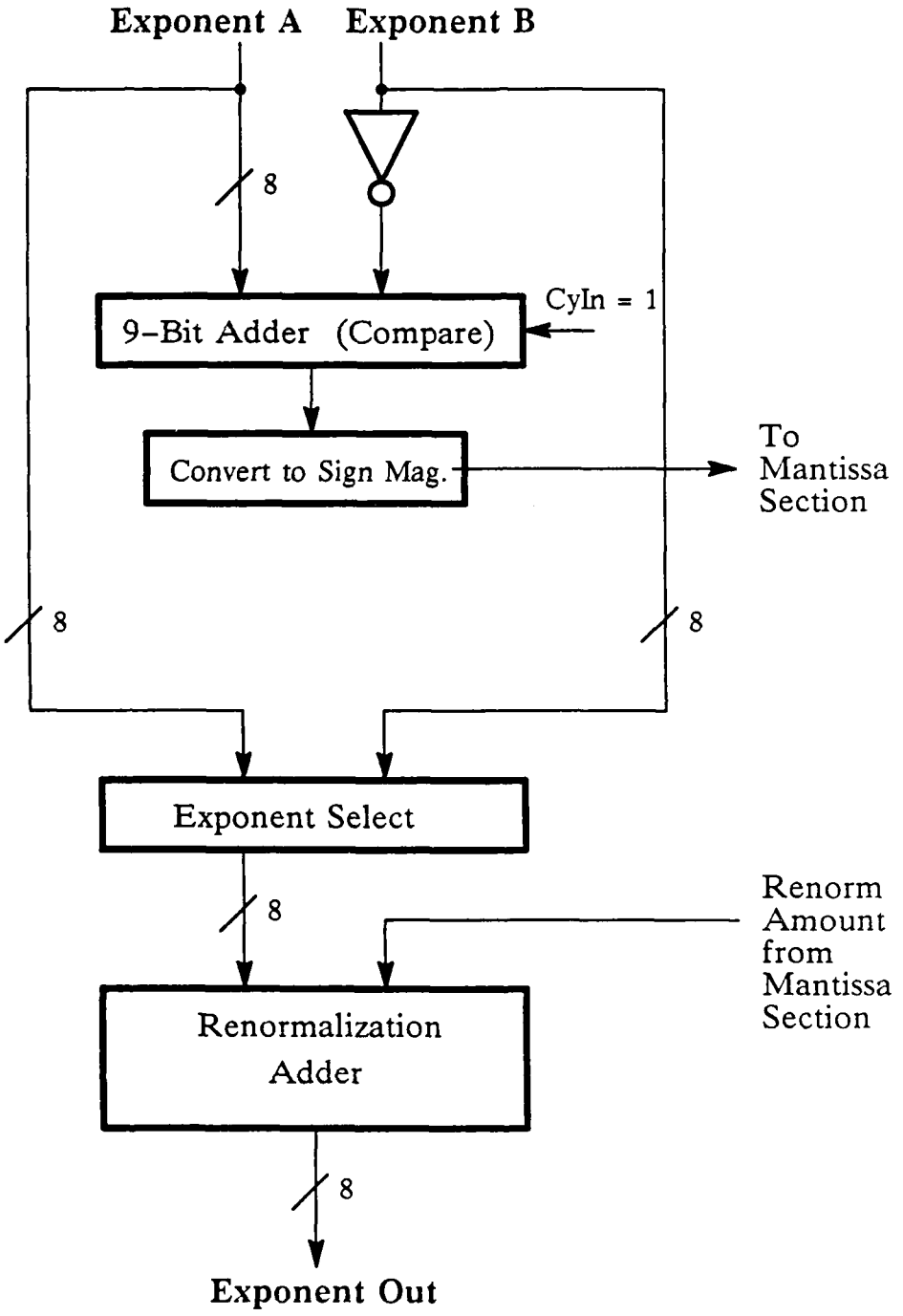


Figure 41. Exponent Calculation for Floating Point Adder

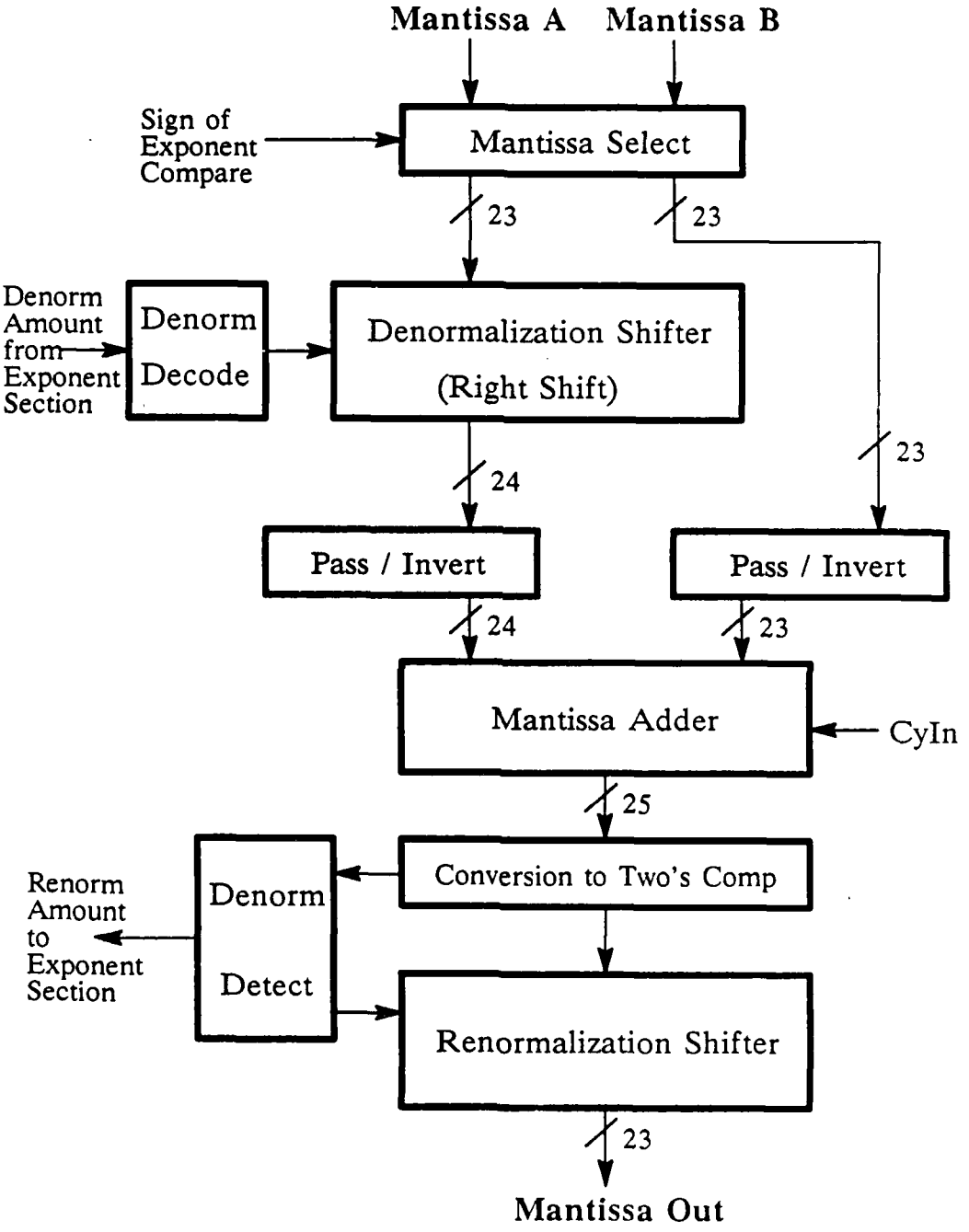


Figure 42. Mantissa Calculation for Floating Point Adder

overflow/leading zero detection, and finally through the exponent renormalization adder. Keys to the adder speed will be fast adders, barrel shifters, and, maybe most importantly, fast encoding/decoding of shift-amount vectors.

Detailed design and implementation of the floating point adder were beyond the scope of this thesis.

4.3.8 Special-Purpose Arithmetic Hardware. It would be inane to assume that all ASP applications could be implemented efficiently with merely an adder and a multiplier. However, it would also be impractical to attempt to provide a full hardware implementation to all possible operations. The ASP architecture provides a simple solution to this problem. Numeric methods, such as the Newton-Raphson approach, have been developed to calculate most conceivable operations using convergence algorithms. For an in-depth analysis at convergence algorithms, see [Bai87]. A common characteristic of all convergence algorithms is that they converge much more rapidly in the initial "guess" is close to the actual solution. For example, an algorithm which calculates a result with 32-bit precision might require only 3-4 iterations to converge from a 4-bit "seed", but might require 8-10 iterations to converge if no seed is provided. In fact, many algorithms require that the initial guess be within certain limits for the algorithm to converge.

The ASP architecture supports these numeric methods by providing allowing a seed to be inserted into the datapath. The seed is gated onto the datapath in a manner similar to literal insertion from the XROM. In this case, the addressing for the seed is received from the input data which is driven onto the A Bus. For example if the operation "$1/x$" is required, "x" is driven onto the A Bus and acts as the address of a ROM or PLA which contains the initial guess at "$1/x$". The output of the lookup table ROM/PLA

is gated onto the C Bus, from which it can be stored in any register. The size of the lookup table (both number of words and the number of bits to be inserted) is determined by analysis of the convergence algorithm. Simulation of the algorithm in a HOL will determine the number of iterations required based on various seed sizes and accuracy required.

Once the size of the lookup table is known, the table can easily be implemented using a ROM or PLA. In general, a PLA is only practical for small lookup tables (32 words or less). The PLA can quickly be laid out using the standard PLA cells from the AFIT library. For larger tables, the XROM Optimizer can be used to automatically lay out the table.

CHAPTER 5

VLSI Implementation

5.1 Introduction

In order to verify the macrocells that were developed for the ASP design library, a prototype ASP design should to be fabricated. This prototype would verify not only the functional correctness of the macrocells, but will also provide data on the maximum clock frequency that the hardware will support. Two approaches were considered for the prototype ASP: applying the ASP architecture to an actual application or building a chip with the microcode optimized for testing. Since the primary purpose of the prototype ASP was to verify the hardware, the decision was made to implement a chip which contained microcode specifically written to exercise the ASP hardware. Implementation of an actual application would likely not provide full microcode capabilities to test all functions.

Several macrocells from the ASP library have already been fabricated. The AFIT XROM is being thoroughly tested with a separate XROM chip which was implemented for the VLSI Design course (EENG695) during the Winter 87 quarter. This chip contains four different XROMs of size 12K, 24K, 48K, and 72K, which contain lookup table data for several mathematical functions. The XROM chip will verify access times for different size XROMs. Additionally, the IEEE floating point multiplier (described in Appendix A) was implemented as a separate chip. It will verify the functional correctness and performance of the multiplier.

Since the multiplier is being tested separately, and the IEEE floating point adder has not been completed, the ASP prototype processor will be an integer-only architecture. The ASP datapath macrocells were designed such that inclusion or exclusion of the floating point macrocells would not impact the operation of other macrocells (other than an increase in capacitance on the data busses). Thus, testing of the ASP architecture without the floating point macrocells will provide an accurate gauge of the ASP capabilities.

The prototype processor is designed for a clock frequency of 20 MHz. Since the chip is using precharged busses, approximately 40ns of the 50ns clock clock cycle is available to drive the data from the registers, through the ALU or shifter, and back to the registers.

5.2 Hardware Implementation

5.2.1 Floorplan. The prototype ASP chip was implemented as a 24-bit integer processor. Figure 43 shows a floorplan diagram of the prototype chip, showing the pinout and approximate relative sizes of the major subsections. The ASP layout was designed to fit in a 64-pin 7900 X 9200 micron package. The pinout of the prototype chip provides the following signals:

8 - Vdd /GND
5 - Clock (PQ1, PQ2, TestPQ1, TestPQ2, Precharge)
24 - Data (bi-directional)
10 - Program Counter output
1 - Branch Condition output
1 - Chip Reset
4 - Flags output
4 - Handshaking to Host
4 - Testability pins
2 - Spare
---
63 - Total Pins (one pin left free for substrate contact)

5.2.2 Control Hardware. The control hardware will vary little between different ASP applications. The prototype control section contains a 13K-bit XROM, organized as 256 words of 52 bits each. Although the prototype only requires an 8-bit program counter and a three-deep subroutine stack, the library microcode sequencer was used, which provides a 10-bit counter and 7-word stack. Only 8 of the 32 possible branch conditions were needed.

5.2.3 Datapath Hardware. The prototype datapath is 24 bits wide. The datapath provides 15 general-purpose registers in the register array, requiring only simple NAND type decoders. A single I/O channel is provided, consisting of an address and a data register. The I/O channel provides 4-line handshaking with a host, but no memory interface was implemented. In order to free pins for testability, the address register output is not driven to output pads. The other source of data to the datapath is provided by the literal insertion hardware. Since it is impractical to insert 24 bits in one cycle, 12 bits can be inserted and the remaining 12 bits can collectively be set to zero or left at one. Arithmetic processing is provided by a barrel shifter and integer ALU. A 24 X 24 barrel shifter is employed, which provides circular shift. The 24-bit integer ALU was easily implemented from library cells, using carry-select stages of 4-4-5-5-6 to provide the required width.

5.2.4 Hardware Design for Testability. The primary goals of design for testability are controllability, testability, and predictability [Fuj85]. Controllability is the ability to drive the inputs of the hardware to a known state. Observability is the ability to "see" the output of the hardware. Predictability is concerned with driving the hardware so that testing can begin at a known state of the hardware.

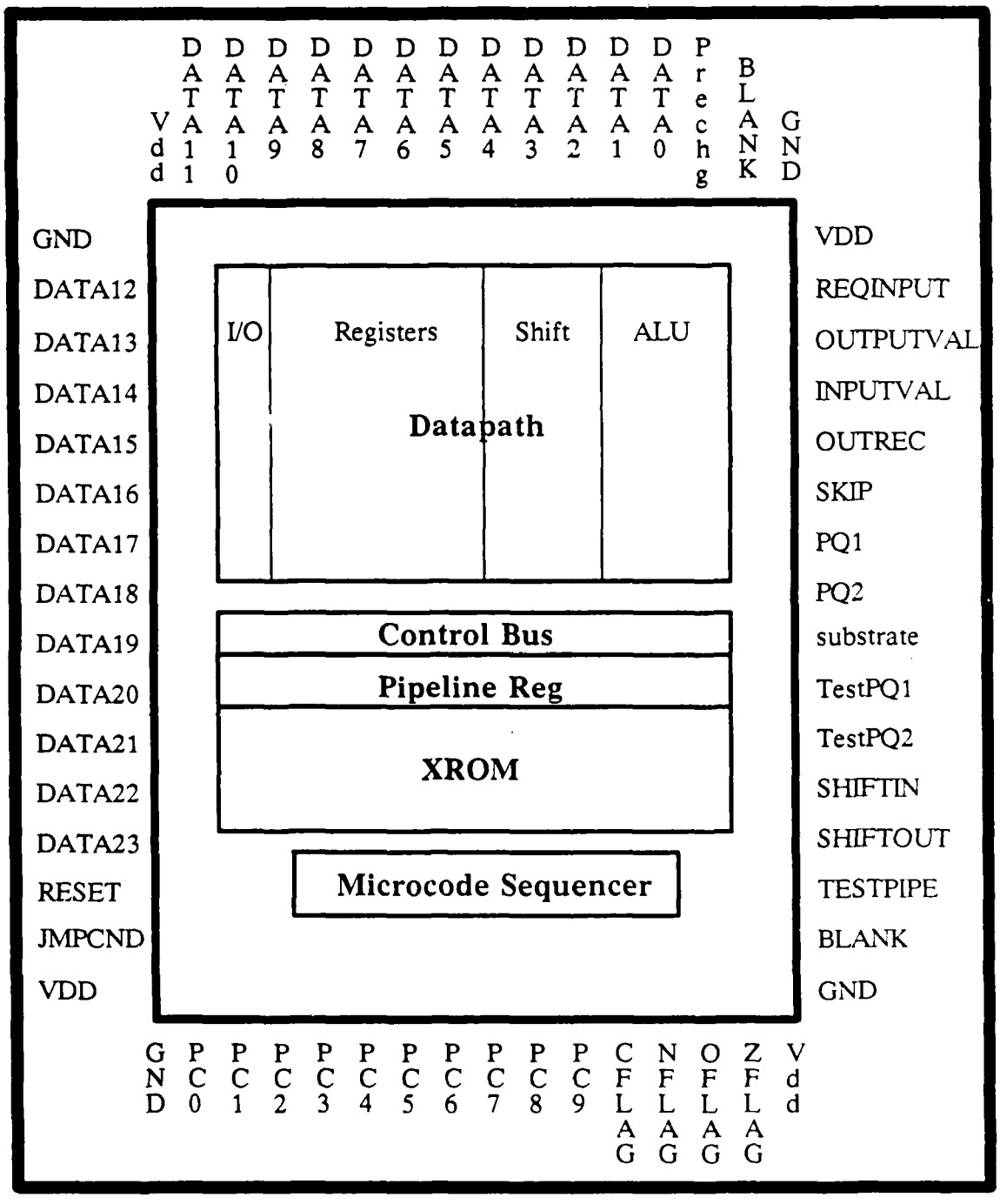


Figure 43. Prototype ASP Floorplan

Predictability is easily provided in the ASP prototype. A master "Reset" input to the chip is provided, which resets the value of the program counter to zero. The microcode is written to begin at Word 0. It is not necessary to reset the datapath registers, since the microcode will not use the value of a register until the register has been loaded. The only other storage circuit that requires care are the flag registers. Since they are not reset upon chip reset, the microcode should not attempt any conditional branches based on these flags until they are set by an arithmetic operation.

Controllability and observability involve not merely the ASP prototype as a whole, but also the ability to test subsections of the architecture. Controllability is provided to the ASP control section via chip reset and the conditional datapath load. Controllability of the control section involves the ability to drive the program counter to a desired value. The chip reset obviously allows the tester to drive the program counter to zero. The conditional datapath load allows the tester to set the program counter to any value desired, by loading this value into the data register, driving it onto the A bus, and then executing a conditional datapath load. The ability to set the program counter to any value allows the designer to write microcode test routines and then cause the hardware to execute them if necessary.

Observability of the control section is provided in several ways. The value of the program counter is driven directly to output pads, so that the tester can easily track the state of the hardware and progress of the microcode. To provide further observability of the microcode sequencer, the "branch condition" output of the branch multiplexer is driven to an output pad. This will allow the tester to better isolate any problems encountered with program branches. The control section also provides observability of the actual control signals which are produced by the XROM. This is accomplished using a

pipeline register which is modified for testability.

The ASP prototype has a 52-bit control word. Since it is impractical to connect each of these control signals to an output for observability, the pipeline register is reconfigured as a shift register. Figure 44 shows the design of the modified pipeline register cell. The MSFF is altered so that it can load from one of two sources, dependent upon the value of the Test pin. When the Test signal is low, the pipeline register continues to function normally, delaying the XROM output one clock cycle. When Test is high, however, the pipeline register loads the value which was previously stored on the adjacent register. In this manner the entire microcode control word can be shifted through the pipeline register. The shift-in and shift-out to the pipe are connected directly to an input and an output pad. By connecting the shift-in pad to the shift-out

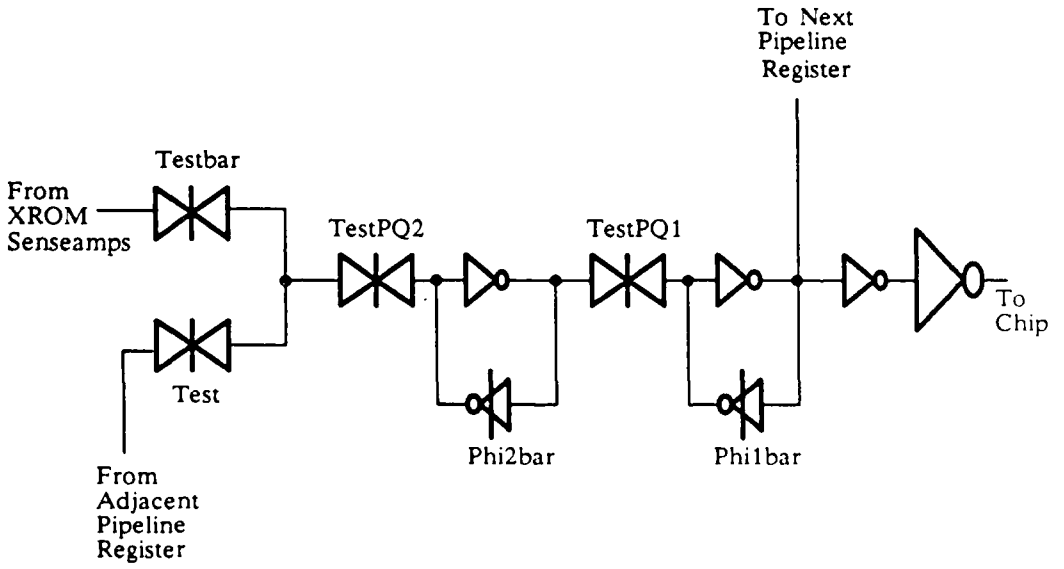


Figure 44. Pipeline Register Modified for Testability

pad, a circular shift can be accomplished, allowing the tester to observe, but not alter, the contents of the pipeline register. Note that the PQ1 and PQ2 signals of the pipeline register must be cycled to shift data through the pipeline. However, if these clock signals are cycling throughout the remainder of the chip, the invalid control signals being shifted along the pipeline will drive the rest of the chip into an invalid state. Thus, the clock signals to the pipeline are provided by the TestPQ1 and TestPQ2 input signals. During normal operation, these signals are same as the clock signals provided to the rest of the chip. During pipeline register shifts, however, only the test clock lines are cycled.

The shiftable pipeline register also provides controllability to the datapath section of the ASP prototype. Instead on connecting the shift-in pad of the pipeline register to the shift-out pad (for a circular shift), a new control word can be shifted into the pipeline via the shift-in pad. The shift-out pad can still be observed to determine the old value of the pipeline register. Controllability over the datapath is further provided by the I/O channel. By loading the data register with the data which he wants tested, the tester can input the data that he wants the microcode routines to process. Observability of the datapath is also primarily provided by the data register. The microcode loads the results of most computations into the data register and drives them to the data pads. Thus, the arithmetic results can be observed directly at the data pads, usually on the clock cycle following the computation. Additional observability of the datapath can also be achieved by observing the result of conditional datapath load microinstructions.

5.3 Microcode Development

For most ASP applications, the microcode is derived from a HOL description of the algorithm. For the prototype chip, however, the microcode is used only to test the hardware. The actual microcode for the ASP prototype is contained in Appendix F.

Figure 45 shows a flowchart of the prototype microcode. The test microcode first tests the datapath. No branch instructions are used during initial datapath testing, so that errors in the microcode sequencer will not prevent datapath testing. First, the ability to load data from the data pads and drive results to the data pads is tested. The data which has been loaded is then used to test each of the arithmetic functions available. In case an error prevents data from being loaded from data pads, the next portion of the microcode loads data into two registers using the literal insertion hardware. The same arithmetic tests are again performed on this new data.

The next portion of the microcode tests the microcode sequencer. Conditional and unconditional jumps, calls, and returns are tested. Nested subroutines are also tested. Finally, the microcode tests the conditional datapath load. Using this capability, the tester can repeatedly load a branch address and perform the arithmetic function located at that address. This allows the ASP to perform similar to a CISC architecture, since the addresses the tester loads cause arithmetic microcode routines to be performed. The provided microcode routines allow normal ALU functions, as well as a software multiply routine.

After the microcode was written, the next step was to create a translation table to interface the microcode to Lt. Hauser's microcode assembler [Hau87]. Appendix D shows the fields which compose the 52-bit microcode word used by the prototype ASP. These fields are easily recast into the format of the translation table required by the assembler (Appendix E). Next, the assembler was executed, producing an integer file which was input into the XROM Optimizer, which produced a Magic-format description of the XROM.

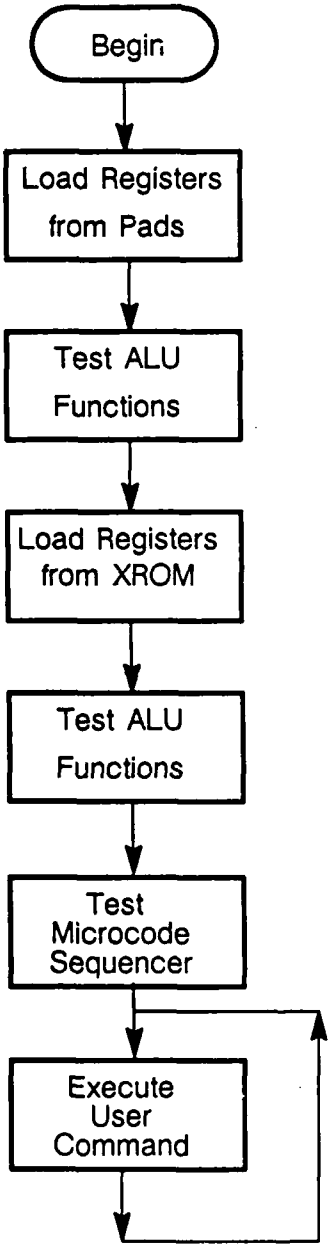


Figure 45. Flowchart of Microcode for the Prototype ASP

5.4 Design Verification

The first step in verifying the design was to ensure the circuit was laid out in accordance with established design rules. Magic's built-in design rule checker (DRC) capability makes this step trivial. Most errors were caught immediately during layout. Any unnoticed errors (perhaps generated by automatic layout tools such as the XROM Optimizer) can be detected by asking Magic to re-check the entire circuit. An even quicker way to determine if errors exist in any of the Magic files was to perform the Unix command "fgrep error *.mag" upon the directories containing the design.

The next verification step was to check the design for two insidious errors which Magic does not check for: "floating wells" and zener diodes. If a P-well is not grounded, or an N-well is not tied to Vdd, the circuit will not work properly. Likewise, if a "well contact" is shorted to a node which is supposed to contain data, a zener diode is formed and the circuit will not perform as desired. Techniques have been developed to identify problems of these types, using Unix script files and a modified Magic technology file.

After design errors have been eliminated, the next step was to output the circuit description from Magic into the CalTech Intermediate Form (CIF), which, like Magic, describes the circuit as a series of rectangles on different layers. The circuit extraction tool, Mextra (Manhattan Circuit Extractor), can then process this CIF file to create a transistor-level description of the circuit. Mextra outputs the circuit description in the SIM format, which is a listing of the transistors and the capacitance of the nodes in the circuit. Additionally, Mextra outputs several other files which are very useful in debugging the circuit: the Nodes file, the Alias file, and the Log file. The Nodes file contains a listing of all of the nodes in the circuit. The Alias file provides a list of all nodes in the circuit which are logically shorted together. The infamous "GND-Vdd" short is a sure

indication that the layout contains errors. The Log file provides a list of labeling inconsistencies which Mextra discovered during extraction. For example, the Log file identifies any label which is attached to two nodes which are not logically connected. This is often a sign that the connection of the two nodes was overlooked. Prior to any further circuit verification, all entries in the Log and Alias files must be either explained or corrected.

The next verification tool applied was Cstat. This tool inputs the circuit description in the SIM format and performs connectivity checks upon the circuit. The most useful output of Cstat is a list of all nodes in the circuit which cannot be set to 0/1, whose value cannot be effected by the inputs, or whose value cannot effect the outputs. Obviously, the designer has to question the function of any node which cannot effect the output of the circuit. This tool often identifies entire sections of the circuit which have been hooked up improperly.

When all Cstat anomalies have been explained, *Stove* can be run on the SIM file. Stove, which was developed at AFIT, extracts higher-level circuit elements from the SIM file's transistor description. For example, Stove identifies inverters, t-gates, and logic gates (AND, OR, NAND, NOR, etc). The output of Stove can be useful in several ways. First, if the actual count of structures extracted does not correspond to the number designed, a problem may exist. Second, Stove provides a listing of all of the transistors which it failed to extract to a higher structure. Examination of this listing often reveals layout errors. Finally, Stove uses the Nofeed and Fixrom tools to produce the FIX file, a modified description of the circuit in the SIM format which is more compatible with Esim, the switch-level simulator. A detailed description of Stove is located at Appendix B.

Nofeed and Fixrom are routines which correct deficiencies in Esim, allowing it to correctly simulate the ASP architecture. Nofeed identifies and removes the feedback loops from MSFFs, converting them to dynamic flip-flops. Although static flip-flops are desirable for the actual circuit, Esim cannot simulate them properly. Fixrom modifies two portions of the XROM circuitry to make it compatible with Esim:

1) The main XROM storage cell is modified to eliminate "fighting" on the drain.

2) A static pulldown inverter in the senseamps is converted into a standard CMOS inverter which Esim is able to simulate.

See Appendix B for a further description of Nofeed and Fixrom.

Once the FIX file has been obtained, the circuit can then be simulated using Esim. The entire ASP chip was simulated "pad-to-pad", which means that inputs to the simulation were only made at the input pads and the simulation results were examined at the outputs. Simulation using Esim is an iterative process, in which simulation is performed, the circuit corrected, the CAD tools are rerun, and the process is repeated until the correct results are obtained. The ASP prototype was completely verified with Esim. The entire microcode was stepped through, one clock cycle at a time, verifying proper performance and yielding high confidence in the circuit which was fabricated.

The process described above for the ASP prototype was also followed to verify the design of the floating point multiplier. The multiplier was simulated "pad-to-pad" prior to fabrication and showed accurate performance.

5.5 Fabrication

The completed ASP prototype chip will be fabricated using a 3 micron dual-metal P-well CMOS process. For an explanation of this process, refer to [Wes85]. Fabrication will be provided through the MOS Implementation Service (MOSIS). MOSIS acts as an interface between designers at universities and the actual fabrication facilities [Fre86]. The chip fabrication was funded by the Defense Advanced Research Projects Agency (DARPA). MOSIS requires that the design be submitted in the CIF format. Communication with MOSIS and submittal of the CIF description are accomplished via Arpanet.

The IEEE floating point multiplier (described in Appendix A) was also fabricated through MOSIS. It used a 2 micron N-well CMOS process. Fabrication of the multiplier required approximately 12 weeks.

5.6 Testing

5.6.1 ASP Prototype Testing. The ASP prototype chip has been completed, but has not yet been fabricated. Testing of the chip will be accomplished as part of a follow-on thesis or as part of the EE795 course during the Fall 88 quarter. This document, other circuit notes, and the Magic layout will be available to aid testing.

5.6.2 Multiplier Testing. The floating point multiplier chip has been received from MOSIS and is currently undergoing functional, performance, and parametric testing. The primary test goals are to verify the functionality of the multiplier and to determine its maximum operating frequency (minimum multiply time). Final testing results on the floating point multiplier were not available at the time of this writing, but will be included in [Jon87].

CHAPTER 6

Application of the Rapid Prototyping Methodology

6.1 Introduction

The preceding chapters have described the design of a general-purpose architecture which can be modified to solve a wide variety of problems. This chapter outlines the methodology to rapidly prototype an application specific processor using this general-purpose architecture. The methodology description is followed by case studies of three applications to which this methodology has been applied.

6.2 The Rapid Prototyping Methodology

Customizing the general-purpose ASP architecture to a particular application requires selection of the proper macrocells from the ASP cell library and development of a customized microcode ROM. The rapid prototyping methodology requires the following steps:

1) The first step in the rapid prototyping methodology is to fully define the given application in algorithmic form. The algorithm to solve the problem should be described in terms of some high-order language (HOL). An excellent language to use in describing the problem algorithm would be the VLSI/VHSIC Hardware Description Language (VHDL). VHDL can be used to describe both the hardware of the ASP and the algorithm which will be stored in the microcode. During this phase of ASP development, a behavioral description of the algorithm is developed. VHDL will be used throughout the ASP design cycle.

2) The next step in the rapid prototyping methodology is to verify that the defined algorithm will perform as required. The VHDL behavioral description of the algorithm should be simulated to verify that the algorithm is correct. Since this behavioral description will serve as the model for microcode development, it is critical that the VHDL model be accurate.

3) Once the application has been described algorithmically, the next step is to analyze the computational requirements of the algorithm. The goal in this step is to determine what types of operations are required by the algorithm, so that the appropriate hardware can be applied to provide an efficient, high-performance solution. The designer should determine the exact operations that the algorithm requires and their frequency of occurrence. He should examine in particular the iterative structure of the algorithm. Instructions which occur outside the iterative structure are executed only once. Conversely, instructions *located within* nested subroutines will be executed repetitively.

The frequency of occurrence of a particular operation will determine the approach to providing the operation in the ASP. The more frequent an operation is used, the more hardware support will be provided. For example, if the "floating point divide" operation is required by the algorithm, but it is only executed a few times, the operation might best be provided by a microcode subroutine. If the divide is required often, a lookup table and convergence algorithm might be provided. If the divide operation is fundamental to the algorithm, a full hardware divider might be employed.

Additionally, examination of the algorithm will provide basic information to the designer. The accuracy requirements and the format of the input data will determine

the required width of the data word. The subroutine depth of the algorithm will approximate the depth of subroutine stack which will be required by the control section of the hardware. The overall amount of code involved in the algorithm will indicate the approximate length of microcode, so that the width of the micro-program counter can ge determined. The I/O requirements of the algorithm may indicate that addition of a second I/O path will increase the ASP performance. In this manner, the nature of the algorithm is mapped into the hardware requirements for high-performance solution of the problem.

4) From the requirements identified through algorithm analysis, the ASP hardware can be fully specified. This step primarily involved selection of the proper macrocells from the ASP library. Unique hardware requirements may require the modification of existing macrocells or the limited design of new macrocells. A VHDL architectural description should be developed for any new or modified macrocell (the macrocell library should include a VHDL description of each macrocell). The ASP hardware can then be specified using a VHDL architectural description of the interface between the required macrocells.

5) Once the required hardware has been specified, the control signals necessary to drive that hardware can be determined. The combination of these required control signals will form the microword for this particular ASP application. In defining the microword, care should be used in the ordering of the fields within the microword. Since the microcode assembler requires that the ordering of the mnemonics used in the microcode be the same as the ordering in the microword, the microcode which will be developed can be made more readable by careful ordering of the microword

fields.

Additionally during this step, the mnemonics for each microword field are defined. The purpose of this step is to create a translation table which will be used by the microcode assembler to translate microcode mnemonics into binary representations of each microword field. Again, careful choice of the mnemonics can greatly enhance the readability of the microcode. A default value for each field within the microword is also defined, to tell the assembler what value to assign to a field if no mnemonic for this field as present in the microcode. For a description of the exact format required for the translation table, see [Hau87].

6) The VHDL behavioral description can now be translated into microcode. To minimize the risk of translation error, the hierarchical structure of the microcode should closely follow the structure of the VHDL algorithm, including most subroutine calls and iterative constructs. Standard subroutines should be used to provide operations not executable in a single instruction, such as divide or trigonometric operations. Due to the time overhead required for subroutines, however, care should be exercised not to overuse subroutines unnecessarily. The manual method now employed for microcode translation is a tedious, error-laden process. Although no software tools are currently available to perform this step, translation from VHDL into microcode is an excellent application for an automated tool to significantly enhance the rapid prototyping methodology.

7) The completed microcode can now be assembled using the microcode assembler developed by Lt Hauser [Hau87]. The assembler requires two input files: the translation table which defines the meaning of the mnemonics used; and the microcode itself

The microcode assembler provides error-checking of the microcode during assembly. It produces several useful output files. The first is the assembler reference file, which provides a listing of the microcode and the value of the actual microword fields which were assembled from the microcode. This file is useful for locating any format errors in the microcode and to check that the translation table is properly defined. The second output file produced by the assembler is the "Addresses" file, which contains a binary representation of the microcode in a format compatible for input into the XROM Optimizer tool. A third output from the assembler is a VHDL description of the XROM which will be developed from the assembler microcode. This VHDL description will be used during the next phase of the rapid prototyping methodology.

8) The required hardware to provide the ASP solution to the given application is now fully defined. The VHDL description of the XROM developed during the previous step can be combined with the macrocell descriptions specified earlier, to form a complete VHDL architectural description of the hardware design. This proposed hardware solution can now be simulated using the same input vectors previously used to verify the original behavioral description of the algorithm. This simulation will verify that the proposed design will properly meet the application's requirements. This step will reveal any errors made in translating the algorithm into microcode. An iterative process of correcting the microcode, assembling it, and then simulating it in VHDL can be employed until the design simulates properly.

9) The verified microcode is next input into the XROM Optimizer. The XROM Optimizer will produce a implementation of the ASP microcode in an XROM microcode store. The Optimizer provides a complete description of the XROM, in the

Magic format. The Magic files provide a hierarchical description of the XROM, with the main storage arrays containing a very large number of cell instantiations. Since Magic has difficulty with the large number of cell instantiations, Magic's performance can be enhanced by "flattening" the Magic description of these arrays to eliminate the cell instantiations. Also, due to the timestamp which Magic employs for each of its files, the entire XROM may require DRC (design-rule checking).

10) The entire design can next be implemented in Magic. Implementation will primarily involve the placement and routing of the macrocells which were previously specified. New CAD tools being developed at the University of California at Berkeley may soon automate this placement and routing task.

11) The implemented design is then verified using the switch-level verification process described in Chapter 5. This verification process is an iterative process of extracting the correcting the implementation, circuit extraction, and then simulating the extracted transistor description using a switch-level simulator such as Esim.

12) The final step in the rapid prototyping methodology is to extract and simulate a VHDL description of the implementation. This is made possible by the Stove tool, which extracts higher-abstraction circuit components from a transistor description of the circuit, and then outputs a VHDL description of the extracted circuit. This VHDL description of the actual circuit implementation can be simulated, using the original vectors developed to simulate the original algorithm.

This effectively closes the design loop, since the same simulation vectors which were used to verify the original specification of the problem were also used to verify that the actual implementation is functionally correct. The implementation is thus logically

equivalent to the specification. Closing the design loop yields a high level of confidence in the reliability of the completed design. VHDL was highly useful throughout the "life-cycle" of the design process. It was used to define the original task, verify that the circuit specification of the solution was correct, and then to verify that the actual implementation was correct.

6.3 ASP Architectures Three ASP architectures have been specified using the general-purpose architecture developed for the rapid prototyping methodology. Two of these architectures have been fully implemented and are ready for fabrication, while the third was specified through microcode implementation.

6.3.1 Prototype Integer Processor. As described in Chapter 5, a prototype integer processor was developed as part of this effort. The prototype processor was developed concurrently with the macrocells which form the ASP architecture library. Design and implementation of this processor required approximately 3 months. A large portion of this design time was due to the fact that the majority of the required macrocells had not been developed prior to the beginning of implementation and had to be custom designed.

At the time of implementation, VHDL descriptions of the ASP macrocells had not been written. Additionally, the VHDL environment at AFIT is still under development. As a result, the full rapid prototyping methodology was not applied to the prototype ASP architecture and VHDL was not integrated into the design process. Nevertheless, the prototype processor was fully verified at the switch level and shown to be functionally correct. The microcode incorporated into the prototype processor was designed to directly test the hardware and did not require translation from a HOL.

6.3.2 PFA Controller. Concurrent with the design of the prototype processor, Lt. Hauser developed the Prime Factor Algorithm (PFA) controller for the Winograd Fourier Transform (WFT) project at AFIT [Hau87]. The PFA controller architecture is responsible for controlling the three WFT processors and their associated memories, as well as interfacing to the host processor. The system specifications require that the chip operate at 20 MHz. The PFA controller is primarily responsible for monitoring status of the WFT system hardware and responding to any discrepancies in their operation. It requires integer operations on 16-bit numbers.

The PFA was designed using the rapid prototyping methodology. Like the prototype processor, the PFA controller effort was unable to employ VHDL in the design process. The PFA controller design benefited from the availability of the ASP macrocell library. As a result, even though the complexity of the PFA controller is similar to that of the prototype ASP, the design time for the PFA controller was reduced to two months. The PFA controller implementation was also verified at the switch level using Esim. The verification process for this chip was easier, since most of the macrocells used had already been verified in the prototype ASP.

The available ASP library cells were adequate for the majority of the PFA controller's processing needs. The major exception was the PFA controller's register array. The ASP library provides a general-purpose register array which communicates with a host processor via the I/O registers. The PFA controller, however, operates under direct control from the host processor and the host processor requires the capability of directly reading or writing to several of the PFA controller registers. Lt. Hauser therefore was required to design a custom interface to the register set, which allowed direct host control over the registers. This custom design added two weeks to the design and

implementation of the PFA controller.

The microcode developed for the PFA controller was implemented directly from flowcharts of the controller algorithm, without the use of an intermediate HOL description. Switch-level simulation of the chip revealed several microcoding errors, which would have been discovered earlier had VHDL been available to the design effort. The microword format of the PFA controller was improved over the prototype ASP microword, resulting in more readable microcode. Additionally, Lt. Hauser was able to use more meaningful mnemonics which combined several microword fields into one, resulting in even more readability. Much understanding of the microcode definition process was gained from the development of the PFA controller microcode.

6.3.3 Kalman Filter Processor. The Kalman Filter processor is one of several processors will be needed for the Space Surveillance effort at AFIT. As part of his thesis research, Captain Shand has modelled the Space Surveillance system using Network 2.5, ADAS, and VHDL. The rapid prototyping methodology was applied to the problem of designing the processor which performs Kalman filtering (an algorithm to perform target tracking). Initially, Captain Shand modelled the Kalman Filter processor using ADAS, and then developed a VHDL behavioral description of the Kalman algorithm. Analysis of the VHDL description revealed that the algorithm required 32-bit floating point operations. Floating point addition and multiplication were frequent, but floating point division was also used widely within the inner loop of the algorithm. Further analysis showed, however, that the divisor in the inner loop divisions was always the same. Therefore, the divisor and its inverse $(1/x)$ could be calculated outside the iterative structure. The algorithm could then merely multiply by the inverse during inner loop operations. The inversion operation was seldom needed and could be performed using a

convergence algorithm without seed (i.e. no hardware support). Kalman filtering also requires the square root operation to be performed once during the algorithm. It was also be provided by a software convergence algorithm.

Once the algorithm was analyzed, the hardware could be specified. The standard ASP library macrocells (floating point adder and multiplier) were sufficient to provide all required arithmetic support. From the hardware specification, the microword and microcode format were defined. Finally, the VHDL behavioral description of the Kalman filtering algorithm was translated into microcode.

The application of the rapid prototyping methodology to the Kalman Filter processor was performed as a demonstration during the AFIT Association of Graduates Symposium in October 1987. Prior to the symposium, Captain Shand had completed the VHDL behavioral description of the Kalman algorithm. In a three day period during the symposium, the rapid prototyping methodology was applied to this application to obtain a hardware specification of the Kalman Filter processor and the required microcode to allow the processor to perform Kalman filtering. The ability to rapidly design a custom application specific processor from a VHDL description of the required algorithm was well demonstrated.

6.4 Conclusion

This chapter has described the rapid prototyping methodology developed to apply the general-purpose ASP architecture to a specific application. The methodology relies heavily upon the use of VHDL as a modelling and simulation tool. Extension of the AFIT VHDL environment will greatly facilitate the development of application specific processors. The rapid prototyping methodology has been successfully applied to three

different custom processors, two of which have been fully implemented for fabrication. The first prototype processor required approximately three months, but this time was reduced to two months for design of the PFA controller. The Kalman processor demonstration required only three days to define the processor architecture. As experience was gained in developing custom ASP architectures, the design time for these processors continued to decrease. These successful applications of the methodology demonstrate that the capability now exists to design most custom processor architectures in less than three months.

CHAPTER 7

Conclusions/Recommendations

7.1 Conclusions

Numerous research programs within the Department of Defense require application specific processors to perform computation and provide control. High-performance VLSI solutions are seldom being applied to these applications, however, due to the prohibitive time required for the design and verification of custom VLSI architectures. This effort presents a solution to this problem.

This thesis has described the hardware and software methods required for the rapid prototyping of application specific processors (ASPs). The methodology is based upon the design of a general-purpose processor architecture that can be modified via microcode to solve specific applications. The ASP architecture can be easily adapted to meet various types of applications. The control section can be used in most applications without modification. The XROM is automatically laid out. Each of the macrocells in the datapath is designed using a bit-splice approach, so that the width of the datapath is easily modified. The busses within the datapath are laid out in a regular structure, so that macrocells fit together easily. This allows the easy addition or removal of macrocells to fit any application.

A prototype processor, which contains the majority of the ASP hardware with the exception of the floating point macrocells, has been implemented and fabricated. An IEEE standard floating point multiplier was implemented, fabricated, and tested. The XROM Optimizer CAD tool was modified and extended to layout XROMs in the Magic

format. The Stove program was developed to facilitate the verification process and to close the design loop, allowing VHDL simulation of the implemented circuit. A methodology was described for translating a high-order language description of the application algorithm into microcode, which can be assembled by the microcode assembler, optimized, and finally implemented in the XROM.

The methodology for rapid prototyping of application specific processors has been successfully applied to three different applications. These test cases have demonstrated that the rapid prototyping methodology can produce a custom processor implementation in less than three months. The design time for future ASPs using this methodology has thus been dramatically reduced. Using the rapid prototyping methodology, design becomes more a problem of specifying an algorithm than a problem of designing hardware. Transforming ideas into implementations has become a much simpler process.

Rapid prototyping of ASPs can reap rewards within education and throughout the DoD. Ideas, which have in the past required several years to realize, can now be implemented and tested within a single thesis cycle. Rapid prototyping can have a large impact within the DoD toward the insertion of VLSI/VHSIC hardware into new projects. The feasibility of an ASP architecture can be demonstrated quickly, so that project managers will be more inclined toward a VLSI/VHSIC approach, rather than accepting the degraded performance of the off-the-shelf approach.

7.2 Recommendations

7.2.1 ASP Library. A major portion of this effort has been the establishment of an ASP cell library. For rapid prototyping to be possible in the future, this cell library must be maintained. Numerous difficulties were encountered during this effort in

attempting to use existing AFIT library cells, due to the lack of maintenance and documentation. Procedures must be established for modifying or adding to the ASP library. Most importantly, an individual should be designated to the library manager, responsible for its proper maintenance.

7.2.2 Microcode Development Tools. The scope of this thesis did not include the production of any software tools for microcode development. Development of these tools would significantly enhance the rapid prototyping methodology. The microcode assembler developed by Lt. Hauser [Hau87] is an very useful tool, but can be further enhanced to increase flexibility, "user friendliness", and error detection. An automated tool for translating a HOL description of the algorithm directly into a format compatible with the assembler would dramatically reduce the time required for microcode development.

One lesson learned during the microcode development for the prototype ASP was that the ordering of fields within the microword and the mnemonics used within each field had a large impact on the readability of the microcode. In order to make the microcode easy to understand and debug, the mapping of the ASP hardware onto the microcode assembler needs further study. By standardizing the ordering of the microcode fields and using meaningful mnemonics, the microcode would be much more understandable to someone not intimately familiar with the particular architecture.

7.2.3 VHDL Interface to ASPs. There is considerable interest in interfacing VHDL with other VLSI software tools and methodologies. In Chapter 6, the usefulness of VHDL in the rapid prototyping methodology was described. VHDL is useful throughout the design cycle, from initial simulation of the application's behavioral description

through simulation of the implemented circuit. The development of automated tools to further interface VHDL to the design process would be an extremely fruitful area of research. An automated tool could take the VHDL behavioral description and generate the required ASP microcode. This interface could be take even further by developing a silicon compiler which converts VHDL descriptions into completed Magic layouts. This compiler would be responsible for determining which macrocells from the ASP cell library are required, automatically placing these cells, and finally performing the necessary signal routing between macrocells. CAD tools already developed at the University of California at Berkeley can provide the placement and routing aspects of the compiler. The VLSI designer could use VHDL to describe both the hardware and software of the application, and then simulate the performance of the described design. Following successful VHDL simulation, the architecture could then be automatically generated from the VHDL description. The ability to generate layout from a high-level description of the circuit would allow the computer architect with little VLSI design experience to quickly take an idea from initial concept through implementation in silicon. This ability to design a circuit at the highest possible level of abstraction allows creative individuals to stay creative rather than becoming mired in implementation details.

7.2.4 Test Vector Generation.

The use of fault simulation during the development of test vectors needs to be further emphasized at AFIT. Often, the designer merely chooses several general cases to test the hardware, perhaps attempting to set up some particular situation within the circuit. Although the circuit may be fully simulated using these vectors, the functional correctness of the design has not actually been verified.

The fault simulator can determine the actual fault coverage of proposed test set, as well as identify portions of the circuit which are not being thoroughly simulated. Using an iterative process of improving the test set and then determining the resultant fault coverage, the designer can develop test vectors which better evaluate the functionality of the circuit. If total (100%) fault coverage is required, the circuit may require modification to add further testability.

7.2.5 Computer Resources. For rapid prototyping to be effective, computer resources must be readily available. Due to their efficiency as VLSI workstations, the Sun workstations should continue to be dedicated to VLSI design. One or more Suns, with memory and disk upgrades, should be purchased for dedicated VLSI use. It is important that VLSI designers do not have to compete with other applications (such as Interleaf or ADAS) for usage of the dedicated VLSI SUN workstations. Perhaps more importantly, computer resources must be readily available for the "number crunching" associated with circuit extraction and simulation. Severe delays were experienced in the ASP development due to the loading on computer resources. To rapidly implement and verify an ASP architecture, the usage of VLSI-dedicated computer resources (i.e. the ELXSI) needs to be limited to VLSI applications , so that computation-intensive operations can be completed as rapidly as possible.

7.2.6 Prototyping Experts. One consideration involved with the rapid prototyping methodology is the expertise required for its efficient use. A designer who is well familiar with the ASP macrocell library would be able to generate a custom ASP design more rapidly than one with little familiarity. This implies that a full-time designer will be more effective than, perhaps, a thesis student who will use the methodology only once.

A reasonable conclusion to draw is that a central location, responsible for prototyping ASP for DoD agencies, would be more effective using this methodology than a de-centralized approach. Gathering VLSI design expertise to form a VLSI/VHSIC "center of excellence" would provide the continuity and synergy to create an optimal environment for the rapid prototyping of application specific processors.

APPENDIX A

Floating Point Multiplier

A.1 Introduction

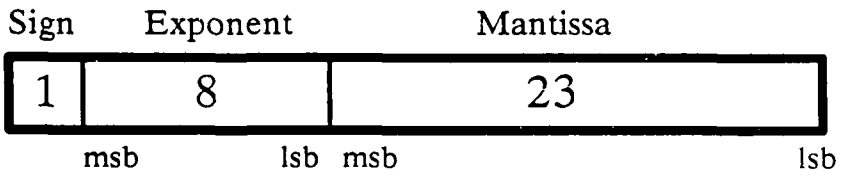
This appendix describes the design, implementation, and verification of an IEEE Standard floating point multiplier. The design approach for the multiplier was to attempt to maximize the speed of the multiply, if necessary at the sacrifice of area, power, and ease of design. The design of the floating point multiplier was initiated as a class project for EE695, VLSI Design. This initial design laid the foundation for the design and implementation completed by this author and Captain Keith Jones. The design of the floating point multiplier logically breaks up into the hardware required for sign, exponent, mantissa, and special case computation.

A.2 IEEE Floating Point Standard

The floating point multiplier was designed to conform to the IEEE Standard for Binary Floating-Point Arithmetic [IEE85]. Figure 46 shows the format of a single-precision floating point number. The 8-bit exponent can store values from -126 to +127. The exponent is biased by adding +127 to the actual exponent value, causing the stored value to always be positive. An exponent of zero is used to indicate a denormalized number when mantissa is not zero or zero if the mantissa equals zero. An exponent value of 255 (all 1's) is used to indicate not-a-number (NaN) or infinity, depending upon the mantissa. The 23-bit mantissa is in unsigned integer format, where the stored mantissa bits makes up the fractional portion of the actual mantissa, in the format: 1.x.

Thus, the floating point multiply requires multiplication of 24-bit mantissas, both of which have a '1' in the most significant bit.

The IEEE Standard requires rounding of the resultant product. Since the multiplication produces a 48-bit product of the mantissas, only the most significant 24 bits are saved. These bits are rounded to the nearest value, unless the bits to be rounded are exactly between two rounding values. In this case, the product is rounded so that the LSB of the remaining bits is zero. The Standard also requires that the hardware trap on certain exception conditions. The multiplier signals underflow, overflow, NaN, and infinity. For a further description of the requirements for rounding and traps, see



Sign: 0 – Positive; 1 – Negative

Exponent: Range –126 to +127; biased +127
Stored value 0 to +255

Exponent zero indicates NaN or zero
Exponent +255 indicates NaN or infinity

Mantissa: Unsigned integer;

Stored as fractional portion of mantissa
with understood 1 in front (1.x)

Figure 46. IEEE Format for Floating Point Numbers

[IEE85].

A.3 Sign and Exponent Computation

Computation of the sign bit for floating point multiplication is trivial. Since the sign bit is only negative if the signs of the two inputs are different, the resultant sign bit is simply the exclusive-or of the two input signs. The computation of the exponent (Figure 47) is somewhat more involved. When two floating point numbers are multiplied, their exponents must be added. An 8-bit adder is thus required to add the two exponents. However, each of the two input exponents is biased by 127, so the desired exponent would be:

exponent_out = exponentA + exponentB - 127

This equation can be easily implemented by adding the two exponents, with a carry-in of 1. The result is then input into a simple subtractor which subtracts 128 from the result.

Since the exponent computation is not on the critical path of the multiplier, a simple carry-propagate adder is used to perform the addition. After the first four bits, the carry signal must be buffered since it has passed through a series of t-gates. The 128-subtractor is implemented using an inverter and a half-subtractor. For the purpose of this discussion, the MSB output of the adder will be called a7 and the carry out of the add will be called a8. To subtract 128 from the output of the adder, 1 must be subtracted from a7. The output, e7, is thus the inversion of a7. The next bit, e8, is equal to a8 if there was no borrow from the previous bit, i.e. if a7 was 1. If there was a borrow, e8 is derived from the inversion of a8. The equation for e8 is thus:

e8 = a7 XNOR a8

The borrow out of the subtractor, e9, is high only if a7 and a8 are low:

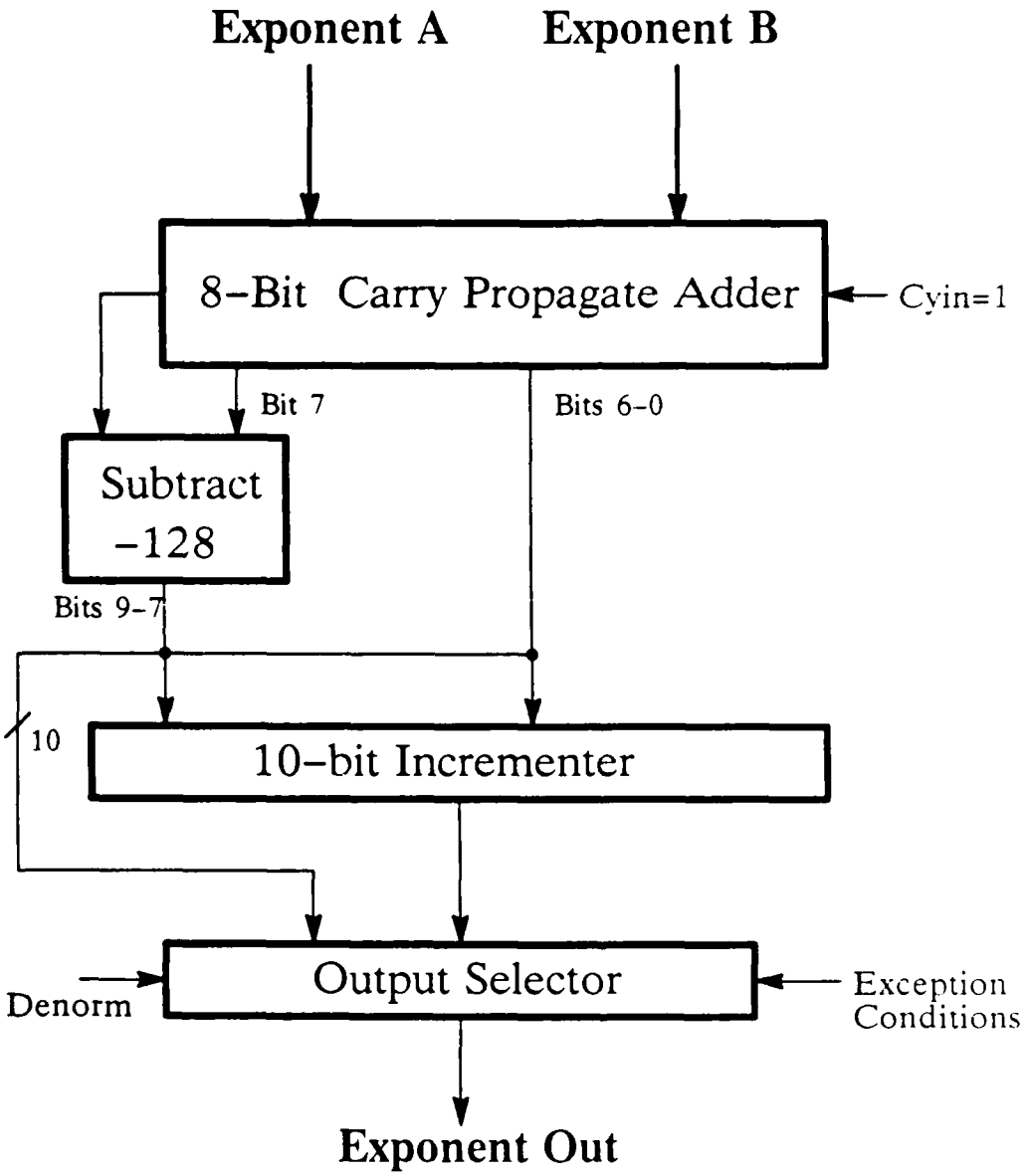


Figure 47. Exponent Section of the Floating Point Multiplier

e9 = a7_bar AND a8_bar

Although e8 and e9 are not actually output from the exponent logic, they are used to form the underflow and overflow signals. Underflow occurs when the addition of the two exponents produces a negative number, since exponents are always supposed to be biased positive. If the subtraction of 128 causes a borrow into E9, underflow has occurred. Similarly, overflow occurs when the addition results in a carry-out into a8 which is not eliminated by the subtract, i.e. a7 and a8 were both '1'. The equations for underflow/overflow are thus:

underflow = e9

overflow = e9_bar AND e8

The 10-bit exponent result must next be incremented again to form the exponent in the case that the mantissa section signals that renormalization is required. The 10-bit incrementer is formed from an inverter and 9 half- adders. Renormalization on the mantissa would require at most a single shift to the right, in which case the incremented exponent would then provide the proper output. When the renormalization signal arrives, a multiplexer selects which of the 10-bit exponents is correct. The correct exponent will then determine which underflow/overflow condition is correct.

Assuming no exceptions are raised, the selected exponent is gated to the output of the multiplier.

A.4 Mantissa Computation

The mantissa computation is the most complex and time-consuming portion of the multiply. The mantissa multiply is a 24 X 24 two's complement integer multiply, so it can serve the dual role as an integer multiplier. The 24 bits of the two inputs are formed

by the 23 mantissa bits and the "understood" leading-edge '1'. Much of the theory for the integer multiply was obtained from a thesis written by Peter Reusens at Cornell [Reu83].

A 24 X 24 multiplier, in its simplest form, is laid out as a 24 X 24 array of adders. The critical path timing delay through this 24 X 24 adder array is the 24 vertical additions, followed by the 24-bit carry propagation along the bottom of the array (Figure 48). Although little can be done about the 24-bit carry along the bottom, techniques have been developed to reduce the effective height of the adder array. The most common technique is the application of Booth's modified algorithm. By encoding three bits of the multiplier at a time, a 24-bit multiplier must be multiplied by the multiplicand only 12 times. The effect is to reduce the height of the adder array to 12. For a further discussion of Booth's encoding, see [Reu83].

The vertical height of the tree was further reduced by structuring the required additions using a Wallace tree structure [Wal64]. The basis for the Wallace tree is that all of the partial product bits are presented to the adder array at the same time. Figure 49 shows how the Wallace tree structure is employed. The partial products for a 6 X 6 multiply are shown. For the column indicated, a05 and a14 would normally be added together first, then their sum would be added to a23 (with a carry from previous column), then with a32, then a41, and finally this sum would be added to a50. Using the Wallace tree approach, a05, a14, and a23 are added together. At the same time, bits a32, a41, and a50 are also being added. The results of these "level 1" adds are then added together by a "level 2" adder, along with a carry from previous column. The result is that the vertical delay through the 6-bit multiplier was reduced from five to three. For the 24 X 24 multiplier, Wallace trees further reduced the height of the adder array from 12 to 6.

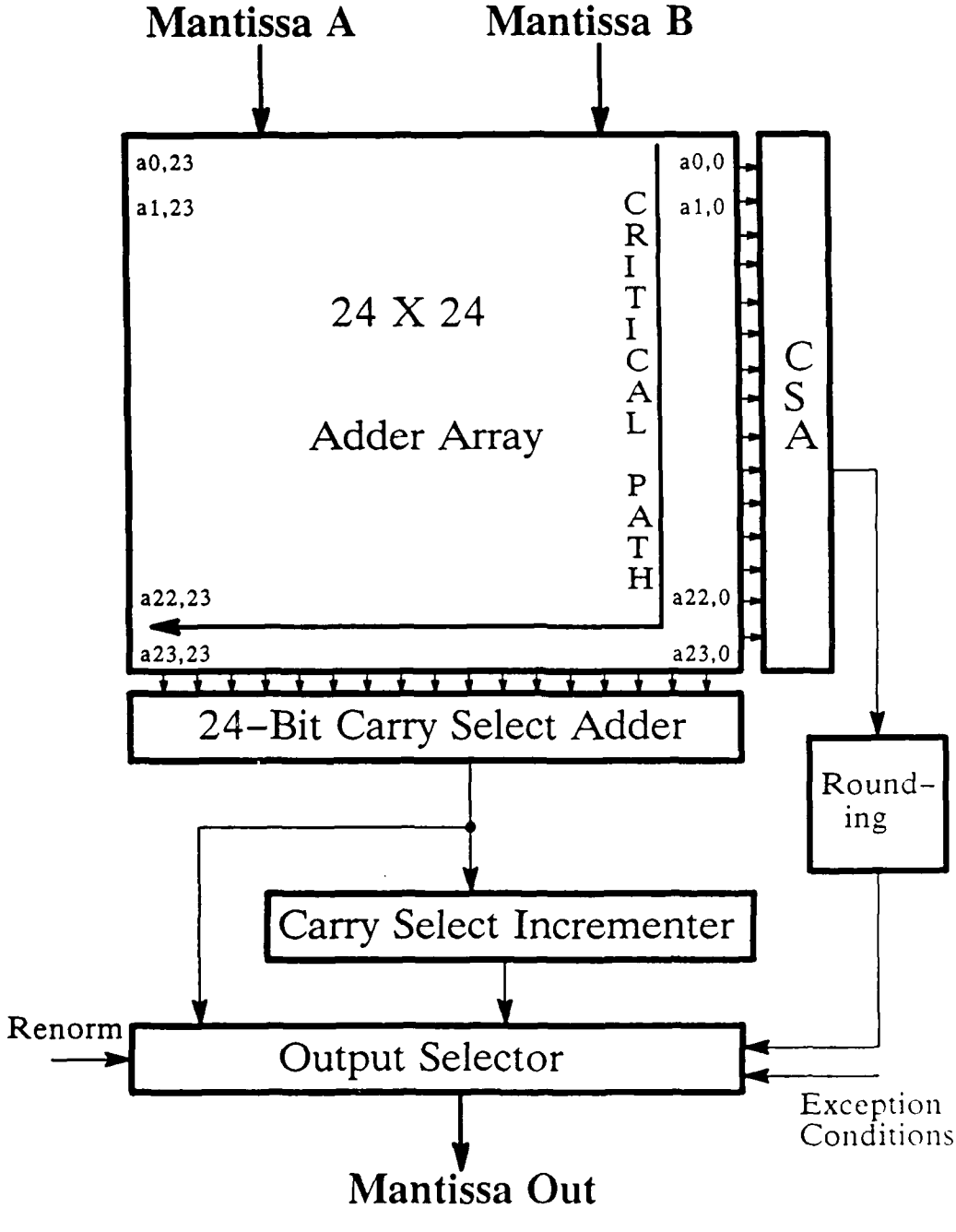
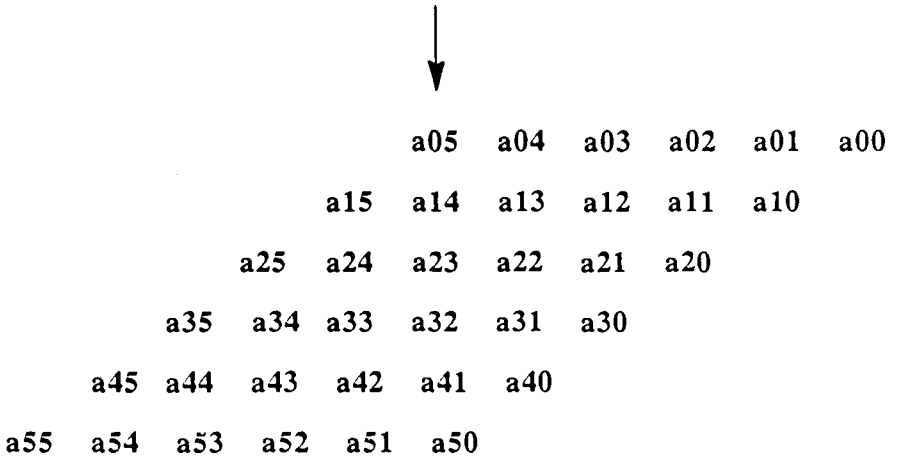


Figure 48. Mantissa Section of the Floating Point Multiplier



Wallace Approach:

First Level Add: Sum1 =a05 + a14 + a23
Sum2 =a32 + a41 + a50

Second Level Add: Sum3 = Sum1 + Sum2 + carry in

Third Level Add: Sum4 = Sum3 + carryin + carryin

Figure 49. The Wallace Tree Approach

The 24 X 24 multiplier was designed using a combination of the Booth's algorithm and the Wallace tree structure. Figure 50 shows the design of the basic adder cell for the adder array. Note that the outputs of the adder have been buffered, due to long routing lines within the array. Special Booth multiplexers provided the partial product bits to the "level 1" adders within the array.

In addition to the basic adder cell, two 24-bit carry-select adders were employed within the adder array. These carry-select adders perform the vertical carry-propagation down through the adder tree and the horizontal carry along the bottom of the array.

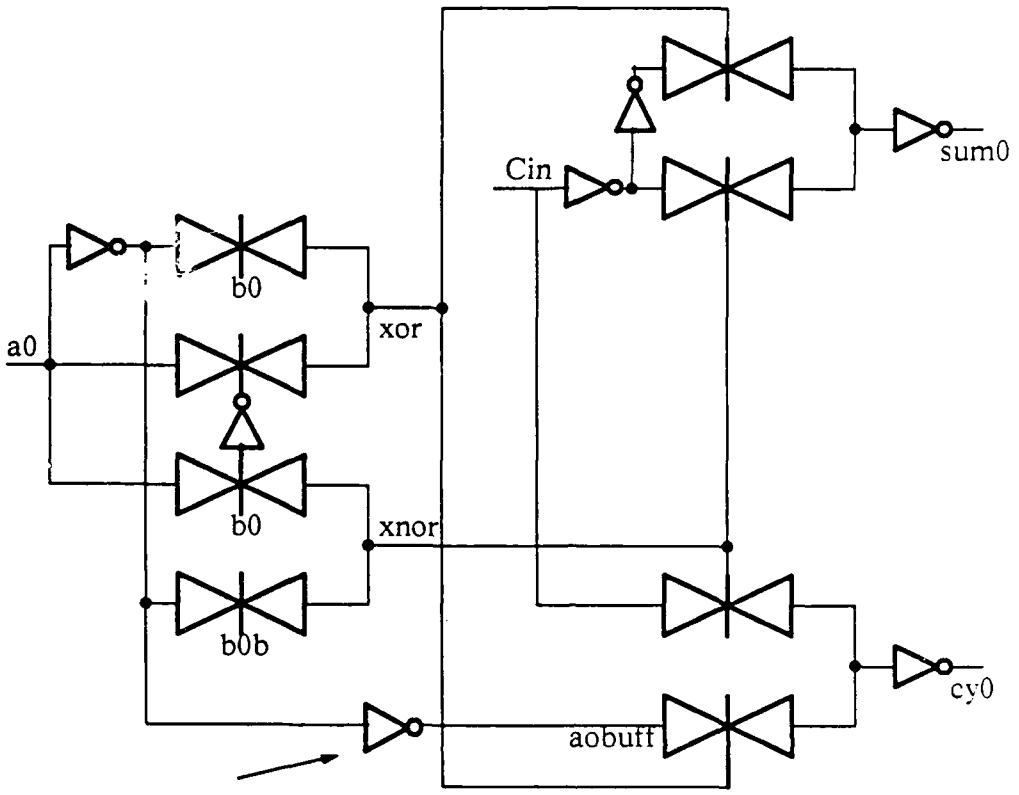


Figure 50. Carry Propagate Adder with Driven Outputs

For a discussion of carry-select adders, see the description of the ALU adder in Chapter 4.

Since the two 24-bit mantissas that were multiplied were in the range 1.0 <= m < 2.0, their product will be in the range 1.0 <= m < 4.0. If the product is 2.0 or greater, it must be renormalized to floating point format via a right-shift. As discussed earlier, this also requires incrementing the exponent. Additionally, the 48-bit product must be rounded. This rounding may require 24 MSBs of the mantissa to be incremented. However, which bits make of the 24 MSBs is also dependent upon whether renormalization

was necessary. The multiplier must, therefore, first determine if renormalization is necessary (a '1' in MSB of product), which in turn determines which bits will actually be rounded. Rounding, if required, can then be accomplished by incrementing the proper 24 bits. Since rounding is on the critical path of the multiply, the incrementer was built as a carry-select incrementer, similar to the carry-select adder. The 23 LSBs of the increment form the product mantissa, unless exceptions were raised elsewhere in the multiplier.

A.5 Special Condition Hardware

The floating point multiplier must detect if certain exception conditions occur. The exponent section is responsible for detecting underflow/overflow. If underflow is detected, it is flagged and the product is set to zero. If overflow occurs, the product is set to infinity. The multiplier must also detect special conditions on the input numbers. Static NAND/NOR gates are used to detect all zeros or ones in the mantissas or exponents. If either input is NaN, the result will be NaN. If either is infinity, the result is infinity. Likewise, a zero input produces a zero result. The multiplier does not support multiplication of denormalized numbers, but instead raises an exception indicating that one of the inputs was a denormalized number. Implementation of these special cases is accomplished using multiplexers at the outputs of the exponent and mantissa sections, which set the outputs to the proper value if a special condition is detected.

A.6 Implementation, Verification, and Fabrication

The floating point multiplier was implemented using the Magic layout tool. Layout of the exponent section was relatively easy. The subcells required for the mantissa section were easily developed, but routing of the signals through the Wallace tree required a

significant investment in time and area. Although the layout of the tree was designed to minimize the required routing, the nature of the tree required some long routing lines, which impacts the performance of the multiplier.

In order to test the functionality and performance of the multiplier, a test chip containing only the multiplier was implemented. A block diagram of this chip is shown is Figure 51. Three registers are used to interface the multiplier to the tester. Two of these registers act as input registers, so that the tester can load the values to be multiplied. This is accomplished bu placing the data on the bi-directional data pads and strobing the LoadA or LoadB input. The result can be read from the third register, by raising the DriveC input, driving the contents of the result register to the data pads. Since the registers are designed as static MSFFs, the loading and reading of registers can be performed at the tester's convenience. The actual multiplication is initiated by the Mult signal, which drives the values contained in the input registers to the multiplier. The Mult signal also allows the output register to load to results of the multiply. The output register is latched on the falling edge of Mult. The actual multiply time can be determined by varying the width of the Mult strobe. The multiply time is the minimum time that the Mult strobe must be maintained high in order to load the correct value into the result register.

After implementation, the multiplier chip design was thoroughly verified using all available tools. The chip was switch-level simulated using Esim and verified to be functionally correct from "pad to pad". The chip was then submitted to MOSIS for fabrication using a 2 micron P-well CMOS process.

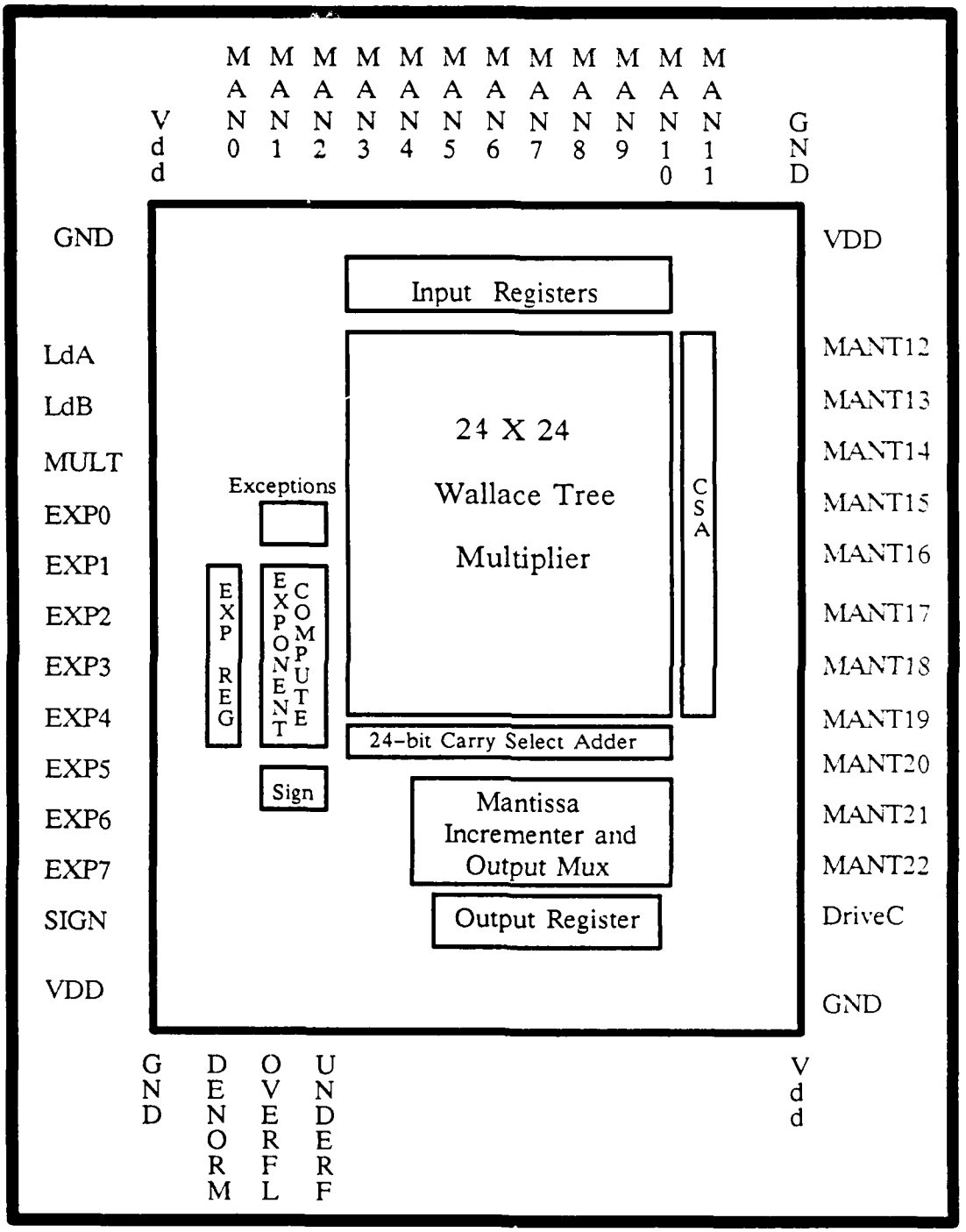


Figure 51. Floorplan or Floating Point Multiplier Chip

A.7 Conclusion

The achieved performance of the multiplier was obtained at the expense of a large investment in time and area. It also resulted in a design with a fairly irregular structure that does not conform to the 81-pitch datapath of the other ASP hardware. Despite the reduction in logical height of the adder array due to the Wallace tree approach, the time savings in required number of adds was somewhat offset by the longer length of the routing lines.

If maximum speed is the driving design parameter, then this approach using the Wallace tree is proper. For applications, however, which do not require optimal speed, the standard Booth encoding approach can result in a significant savings in layout time and complexity. Using the Booth's modified algorithm, the multiplier array would be much more regular and could be adapted to match the pitch of the ASP datapath.

APPENDIX B

Circuit Extraction to VHDL

B.1 Introduction

The rapid prototyping methodology depends heavily upon VHDL. The initial specification of the problem algorithm is accomplished using VHDL. Once the hardware has been specified and the microcode developed, VHDL is again used to demonstrate that the design is correct. Once the designer is convinced that his design will accomplish its purpose, the design must be laid out at the transistor level, hopefully with the aid of some automated placement and routing tool. Once the layout is complete, the designer needs to verify that the transistor-level design will perform the same function as the higher abstraction design. This simulation is usually done with a transistor level simulator, such as Esim.

Simulation at this level, however, requires quite a large investment of both the designer's time and CPU time. It would be much better to run the same test vectors developed for the original VHDL simulation on the completed design. This would effectively close the design loop and result in higher confidence in the completed design. Thus, the need exists for a tool which can extract circuits of a higher level of abstraction from the transistor level and then translate this description into VHDL for simulation.

This appendix describes the tool Stove (.SIM to VHDL extractor), which meets the need for circuit extraction to VHDL. This tool inputs a list of transistors in the .SIM format, extracts the structures which it recognizes, and outputs the higher-abstraction circuit in VHDL format. The program also serves as a front-end for producing circuit

descriptions in the CHIEFS format to provide fault-tolerance testing of the design.

B.2 Software Design

B.2.1 Design Strategy. The design strategy of Stove was to perform circuit extraction of the transistor file in stages, each stage being a higher level of abstraction. At each stage or level of abstraction, the components already extracted can be combined to form components of the next level.

At the lowest level of abstraction, all of the components are transistors. At the next stage, which is called the GATE level, the program extracts inverters, transmission gates (t-gates), and clocked inverters. As the program begins to extract to the third stage, the LOGIC level, it builds its logic components from inverters, t-gates, clocked inverters, and any transistors which were not extracted at the gate level. After each stage, any component which is not extracted is redefined as a component of the next level.

B.2.2 Input Format. The input into the Stove program is a list of transistors in the .SIM format. Each line in the input file contains data on one structure, delimited by spaces. Note that commas cannot be used as delimiters. The first element in the .SIM line is an identifier of what type of data is contained in the line. Stove accepts the following characters as the first element:

p - p-channel transistor

e - n-channel transistor

d - depletion mode transistor

f - transistor identified as "funny" by Mextra

N - nodes

i - inputs

o - outputs

C - capacitance values of the nodes

| - comments

If Stove does not find one of these characters at the beginning of the input line, it will print an error message. After identifying the first character, Stove will process the input line based on the type of data it contains. Lines which contain capacitance data and comments are ignored.

If the input line is a transistor, it must contain at least four and no more than eight elements. The data on the input line must have the following format:

kind gatenode sourcenode drainnode length width Xpos Ypos

The first four elements (kind,gate,source,drain) are required, while the last four are optional. If they are not present, Stove will assume a default length and width of 2 and will set the X and Y position to 0. The "kind" field must be a single character, as specified above. The three node fields must be a character string of less than 100 characters, the first 32 of which must be unique. The last four fields can be either integer or floating point values, but will be rounded to integer values by Stove.

B.2.3 Output Format. Stove provides four different outputs. The first is the status information which is sent to the standard output device. As Stove inputs the data, it will report any format errors in the data. As it processes the input data, Stove prints its extraction results. Stove initially prints the amount of memory required to represent the input data, the number of nodes, and the number of each type of transistor found in the input file. As the program finished extracting a particular pattern, it will

report the number of those patterns found. When Stove completes extraction of a level of abstraction, it reports data on the number of transistors it was unable to extract at that level.

The second output from Stove is a list of the transistors it was unable to extract. When the program has extracted to its highest level of abstraction, any transistors not extracted will be output into the file <stove.out>. This transistor data can be very useful in identifying design errors, since often Stove may have been unable to extract the transistor because of an error during layout. Stove has been successful at discovering numerous design errors in AFIT projects. Stove outputs this transistor data in the .SIM format with 8 fields.

Stove's third output is the file <stove.logic>. This file contains the structures which Stove was able to extract at its highest level of abstraction. This file can be used as the front end of a tool to extract a CHIEFS description (or some other similar description) of the circuit. The format of the data output will depend upon the level of abstraction which Stove currently supports. The current version of Stove extracts to the logic level. The output format at this level consists of seven fields:

kind IN1 IN2 OUT GATE Xpos Ypos

The "kind" field is a single character which identifies the type of component. The following codes are currently being used:

TRANS level
- p - p-channel transistor
- e - n-channel transistor
- d - depletion mode transistor
- f - funny transistor

GATE level
- v - inverters
- t - t-gates
- k - clocked inverters

LOGIC level
- a - AND gates
- o - OR gates
- n - NAND gates (2-input)
- r - NOR gates (2-input)
- x - XOR gates
- y - bad XOR gates
- q - MUXs
- l - resettable NAND gates
- b - bad buffers (an error condition)
- g - 3-input NAND gates
- h - 3-input NOR gates

The next four fields contain the names of nodes connected to structure. IN1 and IN2 are the input nodes, OUT is the output node, and GATE is the node which activates the structure (in case of MUX, the select node). If the nodes are not used, they are set to Vdd. For example, an AND gate does not have a GATE node, so it is set to Vdd. The final two fields contain the approximate X and Y position of the structure. These values reflect the location of one of the transistors which make of the structure.

The final output of the Stove program is the VHDL description itself. The VHDL description provides an entity declaration of the chip, declaring nodes whose names begin with "IZ", "OZ", or "BZ" as inputs, outputs, and bi- directional ports of the circuit. This naming format follows the convention used by several transistor-level VLSI design tools, such as Cstat. Stove also outputs an architectural description of the extracted circuit. This is accomplished by examining the data structure to determine what component types have been found during extraction. These components are declared in the architectural description. Next, a VHDL component instantiation is created for each logic-level structure in the Stove data structure. In this manner, a complete VHDL description of

the circuit is created and stored as an output file.

B.2.4 Data Structure. Stove was designed to run as quickly as possible. Since it must operate on VLSI designs which contain hundreds of thousands of transistors, the data structure must allow rapid search for transistors which match a given pattern. The efficiency of the data structure rather than its size is critical. The data structure used by Stove was derived from the data structure used by Clark Baker of MIT in designing his STAT program.

B.2.4.1 Hash Table. At the transistor level, the data is represented by two basic structures, NODES and TRANS (transistors). To allow rapid access to the nodes, the node names are "hashed" into an integer value, which corresponds to a slot in a hash table array. Each location within the hash table acts as a "hash bucket", which holds all of the nodes whose name hashed to that location. Each location in the table actually consists of a pointer to one node in the "bucket", which is the head of a linked list of nodes contained in the bucket. Each node then contains a pointer to the next node which is contained in the same bucket. The size of Stove's hash array is 11731 buckets.

B.2.4.2 Nodes. Each node is stored as a record which contains 17 fields. Among these fields are NODE pointers to the next node in the bucket and pointers to linked lists of structures to which this node connects. Each node has three TRANS pointers, which point to linked lists of transistors. The "glink" pointer pointers to a list of transistors whose gate is connected to this node. Similarly, the "slink" and "dlink" pointers point to lists of transistors whose source and drain are connected to this node. This data structure allows rapid searching of the data. For example, if Stove is extracting a t- gate, it looks for an e/p transistor pair which share common source and drain

nodes. To determine if a particular node is an input or output of a t-gate, Stove merely searches the source and drain lists of the node, which are normally quite short.

In the same way that it maintains TRANS pointers, each node maintains pointers to GATE and LOGIC level structures, allowing easy search of these structures as Stove extracts higher levels of components.

A final pointer maintained by a node is the "inverts" pointer, which points to a linked list of nodes which are the logical inverse of the node. For example, when Stove finds an inverter, it identifies the input and output nodes as "inverts" of each other. Thus, when Stove later needs to know if two nodes are the logical inverse of each other, as it would when extracting t-gates (whose gates must be the inverse of each other), it merely has to search the node's "inverts" list.

B.2.4.3 Transistors. Transistors are stored by Stove as a record containing 13 fields. The transistor record maintains NODE pointers to its gate, source, and drain nodes. It has TRANS pointers which connect the linked list of transistors which are headed by a node's "glink", "slink", or "dlink" pointers. A transistor record also contains a pointer called "team", which will be used to link the transistor into a list of transistors which compose a higher level structure. The transistor record also maintains data on its kind, size, position, and what higher level structure it is a component of.

B.2.4.4 Gate and Logic Structure. Stove also maintains separate structures for the GATE and LOGIC levels of abstraction. Since these structures are quite similar to the transistor record, they will not be described in detail. Each of these structures maintains a pointer to the lower level components which it is constructed from. These pointers point to the "team" list previously mentioned.

B.2.5 Algorithm. The algorithm used by Stove is straight-forward, composed of the following steps:

1) Loading the data. The input file is read a line at a time. If the line defines an node, input, or output, a new node is created and initialized. If the line defines a new transistor, the transistor is created and initialized. Each of the nodes which connect to the transistor must be created if they do not already exist. These nodes must also be linked to the transistor through the node's glink, slink, or dlink pointers Thus, the data structure is created as each transistor is read in.

2) Extract all desired structures from the transistor level. Inverters, t- gates, and clocked inverters are extracted and defined at the gate level.

3) Re-define all unextracted transistors as gate level structures.

4) Repeat Steps 2 and 3, extracting to the logic level. These steps can repeated numerous times, each time extracting to a higher level of abstraction. At each level, all lower level data structures are preserved and pointers to these lower level structures are created. Thus, an XOR gate has pointers to the t-gates it was derived from, and those t-gates have pointers to the transistors that they were made from.

This strategy allows for both horizontal and vertical growth of the program. As new patterns are defined, they can be added into the existing structure, expanding it horizontally. For example, if a new method of building inverters becomes common, the pattern for this new structure can be added into the program during gate level extraction. The program can grow vertically by extracting to higher abstractions. Stove currently only extracts to the logic level, but the data structure at this level is developed so that it

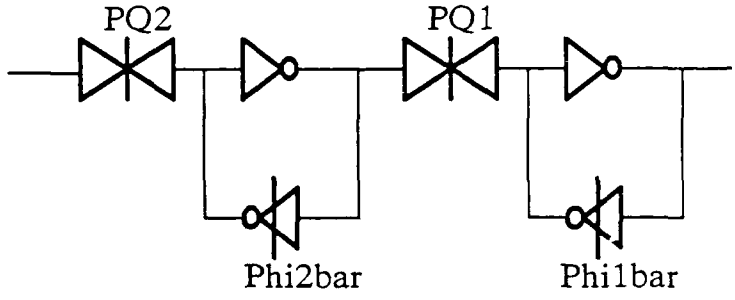
would be easy to begin extraction to the next level.

B.3 Nofeed and Fixrom

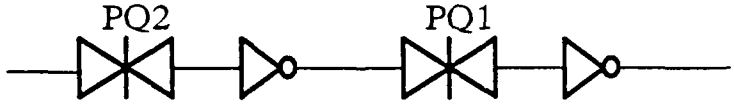
Two variants of Stove were developed to support switch-level simulation. The simulation tool currently used at many universities, Esim, is unable to accurately model certain transistor structures. In particular, it does not properly account for the sizing of the transistors within the circuit. In order to correct this problem, two programs were developed as extensions to Stove. Since Stove already identifies logic structures, it is a simple problem to locate structures which Esim handles improperly and to convert them into a logically equivalent form which Esim can accurately simulate.

The first program, called Nofeed, serves to correct a problem in modelling fully static master-slave flop-flops (MSFFs). Esim can, however, model dynamic flip-flops, which do not have a feedback loop containing a tristate inverter. Nofeed, therefore, locates static MSFFs within the transistor description (.SIM file) and converts them into dynamic flip-flops by eliminating the transistors which form the feedback loop (Figure 52). This is accomplished by marking the transistors to be eliminated. All unmarked transistors are then output into the data file specified by the user. This output results in a .SIM file exactly like the input data file, but without the transistors which created feedback in the MSFFs.

The second program, Fixrom, modifies two portions of the XROM circuitry to make it compatible with Esim. Figure 53 shows the modification made to the SIM description of the XROM storage cell. As discussed in Chapter 4, if two transistors are present in the XROM which are gated by the same wordline and share a common drain, the bitline will be pulled to slightly less than 2.5 volts. Esim does not realize that the senseamp



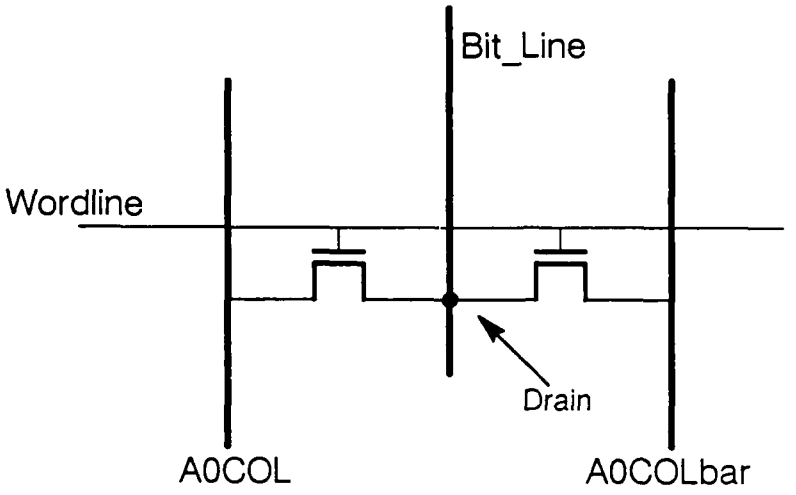
a) Fully Static MSFF



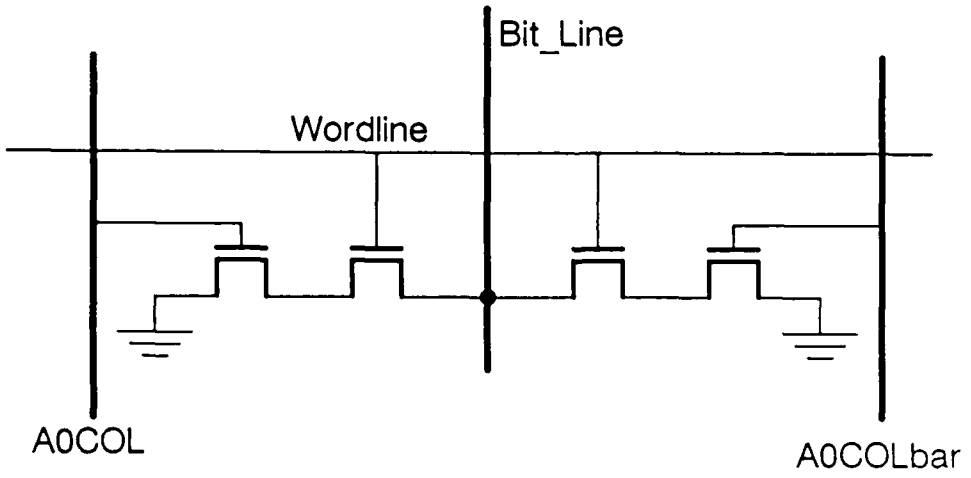
b) MSFF after Nofeed

Figure 52. Removal of MSFF Feedback by Nofeed

detects this a low signal, and thus Esim cannot determine the value of the node. This problem was corrected by adding two extra transistors which eliminate "fighting" on the node. Figure 54 shows the second Fixrom modification. This circuit uses a static pull-down inverter design to minimize capacitance on the bitline. Esim understands static pullups, assuming that they are sized properly, but does not understand static pull-downs. Thus, Esim cannot correctly simulate the inverter. Fixrom replaces the static pulldown with a fully complementary CMOS inverter.

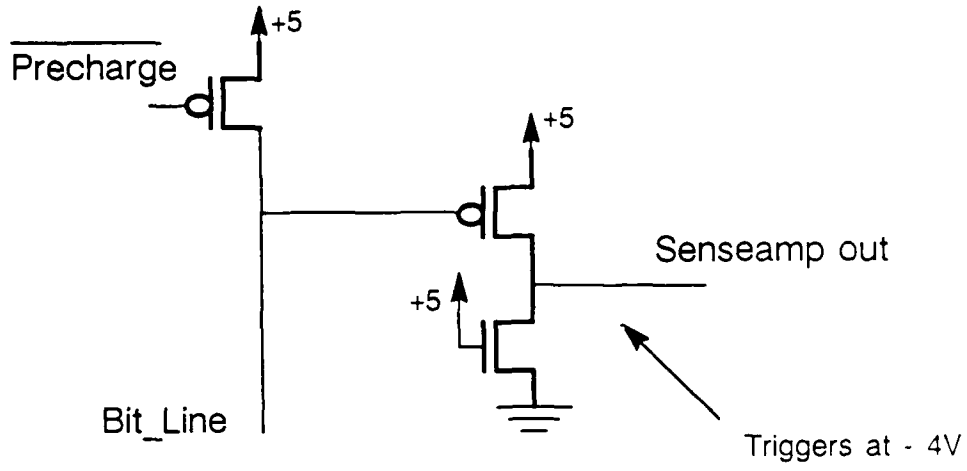


A) Actual XROM Cell

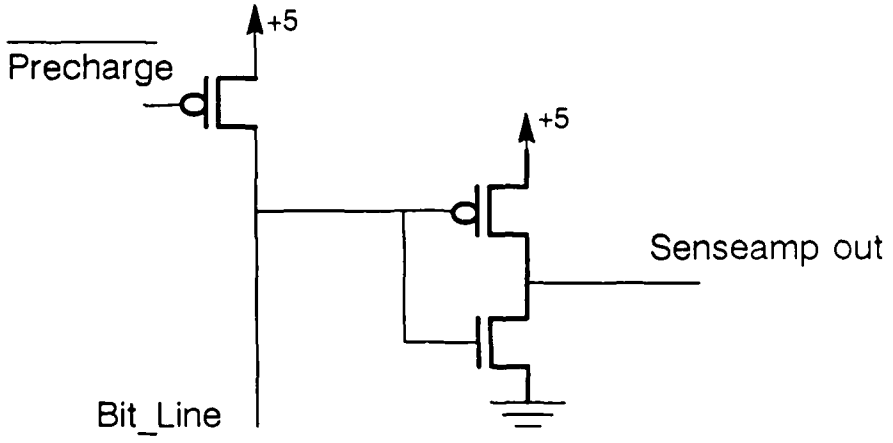


B) Modified XROM Cell

Figure 53. Fixrom Modification of XROM Storage Cell



A) Actual Senseamp



B) Modified Senseamp

Figure 54. Fixrom Modification of XROM Senseamp

B.4 Conclusions

Although Stove is an ongoing project, testing results have shown that Stove is successful at extracting the vast majority of transistors from an average file designed using CMOS pass logic. The data structures and extraction techniques are satisfactory for extracting a higher level circuit description which can be used to generate VHDL or CHIEFS code.

In addition to Stove's capability for circuit extraction, it demonstrated an unexpected capability of error detection within the design. Since Stove extracts by pattern matching to structures which are regularly used in CMOS design, the transistors which it cannot extract are either seldom-used patterns which Stove fails to search for, or an error in the design. Stove can be also used to search for patterns which are not desired. For example, the AFIT VLSI program at one time had a bad XOR cell which had been integrated into several of the designs. By searching for that pattern, Stove would be able to flag any occurrence and then output its location so that the error could be eliminated. Stove's error detection capability may be as nearly important as its extraction capability. Indeed, this area may become the focus of the future development of Stove.

Stove has demonstrated the feasibility of circuit extraction from a transistor list. It is useful as a VLSI design to in discovering layout errors. Most importantly, Stove serves to close the VLSI design loop by providing extraction capability to VHDL. The implemented design can now be simulated in the same high-level language in which it was specified, resulting in high reliability for the fabricated circuit.

APPENDIX C

ASP Microcode Word

Bits 0-2 NAF(3) - Next Address Field
- 0: Continue (Increment)
- 1: Return
- 2: Call
- 3: Branch
- 4: Conditional Datapath Load
- 5: Conditional Return
- 6: Conditional Call
- 7: Conditional Branch

Bit 3 BRO(1) - Branch On - polarity of branch condition
- 0: positive logic
- 1: negative logic

Bits 4-8 CMS(5) - Conditional Mux Select - selects one of 32 branch conditions
- 0: true (unconditional branch)
- 1: zero
- 2: negative
- 3: overflow
- 4: carry
- 5: skip
- 6: inpval
- 7: outrec
- 8-31: not defined

Bits 9-20 Literal Field(12) - Serves as branch address field and literal for datapath insertion.

Bits 21-24 DriveA Field(4) - Selects register which will drive the A Bus
- 0: NOP
- 1-15: Registers 1-15

Bits 25-28 DriveB Field(4) - Selects register which will drive the B Bus
- 0: NOP
- 1-15: Registers 1-15

Bits 29-32 LoadC Field(4) - Selects register which will load from

the C Bus
0: NOP
1-15: Registers 1-15

Bit 33 AddDriveA(1) - Drives Address Register onto A Bus
0: NOP
1: Drive

Bit 34 AddLoadC(1) - Loads Address Register from C Bus
0: NOP
1: Load

Bit 35 DataDriveA(1) - Drives Data Register onto A bus
0: NOP
1: Drive

Bit 36 DataLoadC(1) - Loads Data Register from C Bus
0: NOP
1: Load

Bit 37 DataLoadPads(1) - Loads Data Register from Data Pads
0: NOP
1: Load

Bits 38-42 ShAm(5) - Shift Amount for Barrel Shifter
0: NOP
1-23: Circular shift 1-23 positions to left

Bit 43 LitIns(1) - Insert 12-bit literal into LSBs of datapath
0: NOP
1: Insert literal

Bit 44 Lit0(1) - Set 12 MSBs to 0
0: NOP
1: Pull bits to GND

Bits 45-48 ALU(4) - ALU function select

0: NOP
1: Complement A - C := A'
2: C := A and B
3: C := A xor B
4: C := A or B
5: C := A (move)
6: Set carry flag
7: Reset carry flag
8: C := A + 1
9: C := A - 1
10: C := A + B + cy
11: C := A + B
12: not defined
13: C := A - B
14: C := A - B - borrow
15: Compare A,B (C Bus not affected)

Bit 49 DataReq(1) - Request for input data

0: NOP
1: Data Request

Bit 50 DataValid(1) - Output data is valid at pads

0: NOP
1: Data Valid

Bit 51 Not Defined(1)

APPENDIX D

Translation Table for Microcode Assembler

NAF	000
Ret	001
Call	010
Jmp	011
LoadCond	100
RetCond	101
CallCond	110
JmpCond	111
BRO	0
not	1
CMS	00000
true	00000
zero	00001
neg	00010
over	00011
cy	00100
skip	00101
inpval	00110
outrec	00111
Literal	000000000000
Reg	0000
R0	0000
R1	0001
R2	0010
R3	0011
R4	0100
R5	0101
R6	0110
R7	0111
R8	1000
R9	1001
R10	1010
R11	1011
R12	1100
R13	1101
R14	1110
R15	1111

AddDriveA	0
AddDrA	1
AddLoadC	0
AddLdC	1
DataDriveA	0
DataDrA	1
DataLoadC	0
DataLdC	1
DataLoadPads	0
DataLdPads	1
ShAm	00000
shift1	00001
shift2	00010
shift3	00011
shift4	00100
shift5	00101
shift6	00110
shift7	00111
shift8	01000
shift9	01001
shift10	01010
shift11	01011
shift12	01100
shift13	01101
shift14	01110
shift15	01111
shift16	10000
shift17	10001
shift18	10010
shift19	10011
shift20	10100
shift21	10101
shift22	10110
shift23	10111
LitInsert	0
LitIns	1
LiteralZero	0
LitZero	1

ALU	0000
inv	0001
and	0010
xor	0011
or	0100
mov	0101
setcy	0110
resetcy	0111
incr	1000
decr	1001
adc	1010
add	1011
sub	1101
sbb	1110
comp	1111
DataRequest	0
DataReq	1
DataValid	0
DataVal	1
NotDefined	0

APPENDIX E

ASP Prototype Microcode

```
begin:  nop;                                    the first section of microcode tests
        nop;                                    the functionality of the datapath
        nop;                                    in case microcode sequencer is not
        ;                                       working properly, only sequential
        ;                                       instructions are executed at this point
        DataReq;
        DataLdPads;                             load test data from datapads
        DataVal;
        R0 R0 R1 DataDrA mov;                   put data in R1
        R1 R0 R0 DataLdC mov;                   test ability to put back into datareg
        DataVal;                                output value
        R1 R0 R1 incr;                          test ALU incr function and busses
        R1 R0 R0 DataLdC mov;
        R1 R0 R1 incr DataVal;
        R1 R0 R1 incr;
        R1 R0 R1 DataLdC incr;
        R1 R0 R1 DataLdC decr DataVal;          test decr function
        R1 R0 R0 DataLdC mov DataVal;           test mov
        R1 R0 R1 inv DataVal;                   test complement
        R1 R0 R2 DataLdC mov DataReq;           save data in R2
        DataLdPads;                             get new data
        R0 R0 R3 DataDrA mov DataReq;           put in R3
        DataLdPads;                             get more data
        R0 R0 R4 DataDrA mov;                   and put in R4
        R3 R4 R0 DataLdC and;                   test and function
        R3 R4 R0 DataLdC or DataVal;            test or
        R3 R4 R0 DataLdC xor DataVal;           test xor
        R3 R4 R0 DataLdC add DataVal;           test add
        R3 R4 R0 DataLdC sub DataVal;           test sub
        R3 R4 R0 DataLdC comp DataVal;          test compare
        resetcy DataVal;
        R3 R4 R0 DataLdC adc;                   test adc w/ cy=0
        R3 R4 R0 DataLdC sbb DataVal;           test sbb w/ cy=0
        setcy DataVal;
        R3 R4 R0 DataLdC adc;                   test adc w/ cy=1
        setcy DataVal;
        R3 R4 R0 DataLdC sbb;                   test adc w/ cy=1
        R3 R0 R0 DataLdC shift1 DataVal;        test shift register
        R3 R0 R0 DataLdC shift5 DataVal;
        R3 R0 R0 DataLdC shift15 DataVal;
        R3 R0 R0 DataLdC shift23 DataVal;
```

```
        DataVal;
        ;                                   the next section of code is the same
        ;                                   as the previous, except that a literal
        ;                                   field from XROM inserts data. this
        ;                                   allows testing if I O path is faulty
        #010100011001 R0 R0 R6 DataLdC LitIns LitZero mov;
        R6 R0 R6 shift12 DataVal;
        #001001010101 R0 R0 R7 DataLdC LitIns LitZero mov;
        R6 R7 R3 DataLdC or DataVal;        combine first word in R3
        DataDrA DataLdC incr DataVal;       check alu functions
        DataDrA DataLdC decr DataVal;
        DataDrA DataLdC inv DataVal;
        #010100010101 R0 R0 R6 LitIns LitZero mov DataVal;
        R6 R0 R6 shift12 ;
        #000111000100 R0 R0 R7 LitIns LitZero mov;
        R6 R7 R4 DataLdC or ;               combine second data word in R4
        R3 R4 R0 DataLdC and DataVal;       test alu functions
        R3 R4 R0 DataLdC or DataVal;
        R3 R4 R0 DataLdC xor DataVal;
        R3 R4 R0 DataLdC add DataVal;
        R3 R4 R0 DataLdC sub DataVal;
        R3 R4 R0 DataLdC comp DataVal;
        resetcy DataVal;
        R3 R4 R0 DataLdC adc;
        R3 R4 R0 DataLdC sbb DataVal;
        setcy DataVal;
        R3 R4 R0 DataLdC adc;
        R3 R4 R0 DataLdC sbb DataVal;
        R3 R0 R0 DataLdC shift2 DataVal; test shifter
        R3 R0 R0 DataLdC shift8 DataVal;
        R3 R0 R0 DataLdC shift18 DataVal;
        R3 R0 R0 DataLdC shift22 DataVal;
        ;                                   the next section of the microcode
        ;                                   tests the microcode sequencer
        ;                                   functions
        Jmp A DataVal;                      test unconditional jump
        nop;
        nop;
        DataReq;
A:      DataVal;
        JmpCond skip D;                     skip allows portions of code to be
        ;                                   circumvented (so one bad operation
        ;                                   will not prevent subsequent tests
        nop;
B:      JmpCond not inpval B DataReq;       test external pin control of inpval
        DataReq;
        setcy;
C:      JmpCond not cy C DataReq;
```

```
        DataReq;
D:      nop;
        JmpCond skip E;
        nop;
        Call F;                     test simple subroutine call
        nop;
        CallCond skip F;            test conditional subroutine call
        nop;
        Call G;                     call test for conditional return
        nop;
E:      nop;
        JmpCond skip H;
        nop;
        Call I;                     test nested subroutines
        nop;
H:      nop;
        JmpCond skip M;
        DataReq;
N:      JmpCond not inpval N;
        DataLdPads;                 load value of instruction vector
        LoadCond true DataDrA;      vector to requested location
        nop;
M:      Jmp N;
F:      DataReq;
        DataReq;
        DataReq;
        Ret DataReq;
        nop;
G:      DataReq;
        DataReq;
        DataReq;
        RetCond skip DataReq;
        DataReq;
        Ret DataReq;
        DataReq;
I:      Call J;
        nop;
        Ret;
        nop;
J:      Call K;
        nop;
        Ret;
        nop;
K:      Call L;
        nop;
        Ret;
L:      nop;
        Ret;
```

```
        nop;
        nop;
        nop;
        nop;
        nop;
cmd0:   Call get2 DataReq;                  instruction to load 2 registers
        nop;
        Jmp N DataReq;
get2:   DataReq;
G1:     JmpCond not inpval G1 DataReq;
        DataLdPads DataReq ;                load data
        R0 R0 R1 DataDrA mov ;              put it in R1
        DataReq;
G2:     JmpCond not inpval G2 DataReq;
        DataLdPads DataReq ;                load second data value
        R0 R0 R2 DataDrA mov ;              load it in R2
        Ret;
        DataReq;
add:    Jmp N R1 R2 DataLdC add;            add instruction
        DataVal;
sub:    Jmp N R1 R2 DataLdC sub;            subtract instruction
        DataVal;
and:    Jmp N R1 R2 DataLdC and;            and instruction
        DataVal;
or:     Jmp N R1 R2 DataLdC or;             or instruction
        DataVal;
xor:    Jmp N R1 R2 DataLdC xor;            xor instruction
        DataVal;
shift:  Jmp N R1 R0 R1 DataLdC shift1;      shift instruction
        DataVal;
mult:   R2 R0 R4 mov;                       multiply instruction
        R0 R0 R3 LitIns LitZero mov ;       R3 = 0
        #000000010111 R0 R0 R5 LitIns LitZero mov ; R5 = 23 (loop count)
        #000000000001 R0 R4 R0 LitIns LitZero and ; and to check LSB
        JmpCond zero m1;
        R4 R0 R4 shift23 ;
        R3 R1 R3 add;                       if LSB zero add multiplier
m1:     R3 R0 R3 shift23;                   shift temp result
        #000000000001 R0 R4 R0 LitIns LitZero and;
        JmpCond zero m3;
        R4 R0 R4 shift23 ;
        R3 R1 R3 add;
m3:     R5 R0 R5 decr;                      decr loop count
        JmpCond not zero m1;                if not 0, continue mult
        nop;
        R3 R0 R0 DataLdC shift23;           shift result right 1
        Jmp N DataVal;                      all done; return
        DataVal;
```

```
shR1:  R2 R0 R2 decr;                          shift R1 amount in R2
       JmpCond not zero shR1 R1 R0 R1 shift1;
       R1 DataLdC mov;
       Jmp N DataVal;
       DataVal;
end;
```

BIBLIOGRAPHY

[Bai87] Bailey, Mickey J., "High Speed Transcendental Elementary Function Architecture in Support of the Vector Wave Equation," *MS Thesis, AFIT/GE/ENG/87D-3*, School of Engineering, Air Force Institute of Technology (AU), (to be published December 1987).

[Bie85] Biems, J. L. and others, "The MC68020 32-Bit MPU: Opening New Doors," Reprint from Wescon 1985 Profesional Program Papers, Motorola Bulletin AR232, Motorola Literature Distribution, Phoenix, AZ (1985).

[Cal86] California, University of Berkeley, *Berkeley Distribution of VLSI Design Tools*, Computer Science Division, EECS Department, University of California at Berkeley (1986).

[Col85] Colwell, R. P. and others, "Computers, Complexity, and Controversy," *Computer* **18** pp. 8-19 (September 1985).

[Dia87] Diaz, Juan E. and J. W. DeGroat, "Microcoded RISC Processor for Calculation of the Vector Wave Equation," *Proceedings of the IEEE 1987 Custom Integrated Circuits Conference*, IEEE Press, (March 1987).

[Fre86] French, L. E., "A RISC Controller for the CAM-PUTER System," *M.S. Thesis, AFIT/GE/ENG/86D-53*, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, (December 1986).

[Fuj85] Fujiwara, H., *Logic Testing and Design for Testability,* MIT Press, Cambridge (1985).

[Gal86] Gallagher, David M. and others, "A RISC Optimized for Solving Systems of Linear Equations," EENG 588 Class Project, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH (September 1986).

[Gal86A] Gallagher, David M., "Circuit Extraction from SIM to VHDL," EENG 653 Class Project, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH (December 1986).

[God87] Godfrey, Jim and others, "An IEEE Floating Point Adder," EENG 695 Class Project, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH (March 1987).

[Hau87] Hauser, Robert S., "Design and Implementation of a VLSI Prime Factor Algorithm Processor," *MS Thesis, AFIT/GCE/ENG/87D-5*, School of Engineering, Air Force Institute of Technology (AU), (to be published December 1987).

[Hen84] Hennessey, J. L., "VLSI Processor Architecture," *IEEE Transactions on Computers* **C-33** pp. 1221-1246 (December 1984).

[Hen82] Hennessey, J. L. and others, "The MIPS Machine," *Proceedings of IEEE COMPCON Spring 1982*, pp. 2-7 (February 1982).

[Hen87] Hennessy, John and others, "Research in VLSI Systems," Technical Progress Report. Report to DARPA Conference, Center for Integrated Systems, Stanford University, Stanford, CA (December 1986 - March 1987).

[IEE85] IEEE, Computer Society Standards Committee, "IEEE Standard for Binary Floating-Point Arithmetic," ANSI/IEEE Std 754-1985, IEEE Press, New York (1985).

[Joh87] Johnson, Thomas L., "The RISC / CISC Melting Pot," *Byte* **12** pp. 153-160 (April 1987).

[Jon87] Jones, Keith and David M. Gallagher, "Testing of the Floating Point Multiplier Chip," EENG 795 Class Project, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH (December 1987).

[Kat85] Katevenis, Manolis G. H., *Reduced Instruction Set Computer Architectures for VLSI,* The MIT Press, Cambridge, MA (1985).

[Lin85] Linderman, Richard W. and others, "CSTAT: Sim File Checker for CMOS Chips," AFIT CAD Tool Documentation, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH (July 1985).

[Mac86] MacGregor, D. and others, "The Motorola MC68020," pp. 429-446 in *Tutorial on Advanced Microprocessors and High-Level Language Computer Architectures,* ed. Veljko Milutinovic,IEEE Computer Society Press, Washington DC (1986).

[Man82] Mano, M. Morris, *Computer System Architecture,* Prentice-Hall, Englewood Cliffs, N.J. (1982).

[Mea81] Mead, C. A. and L. Conway, *Introduction to VLSI Sytems,* Addison-Wesley, New York (1981).

[Mye82] Myers, G. J., *Advances in Computer Architecture,* Wiley, New York (1982).

[Ous87] Ousterhout, John, "Editing VLSI Circuits with Caesar," User's Manual, Computer Science Division, EECS Department, University of California at Berkeley (January 1987).

[Prz86] Przybylski, S. A. and others, "Organization and VLSI Implementation of MIPS," pp. 202-240 in *Tutorial on Advanced Microprocessors and High-Level Language Computer Architectures,* ed. Veljko Milutinovic,IEEE Computer Society Press, Washington DC (1986).

[Rad83] Radin, G., "The 801 Minicomputer," *IBM Journal of Research and Development* **27** pp. 237-246 (May 1983).

[Reu83] Reusens, P.P., "High Performance VLSI Digital Signal Processing Architectures and Chip Design," *Ph.D. Thesis Cornell University,* (August 1983).

[Rob87] Robinson, Phillip, "How Much of a RISC?," *Byte* **12** pp. 143-150 (April 1987).

[Ros87] Rossbach, P. C., R. W. Linderman, and D. M. Gallagher, "An Optimizing XROM Silicon Compiler," *Proc. of the IEEE Custom Integrated Circuits Conference,* pp. 13-16 (1987).

[Ros85] Rossbach, Paul C., "Control Circuitry for High Speed VLSI Winograd Fourier Transform Processors," *MS Thesis,GE/ENG/85D-95,* School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH. (December 1985).

[Sed82] Sedra, Adel S. and Kenneth C. Smith, *Micro-Electronic Circuits*, Holt, Rinehart, and Winston, New York (1982).

[She86] Shephard, Carl G., "Integration and Design for Testability of a High Speed Winograd Fourier Transform Processor," *M.S. Thesis, AFIT/GE/ENG/86D-46*, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, (December 1986).

[Sil86] Silbey, A. and others, "A survey of Advanced Microprocessors and HLL Computer Architecture," *Computer* **19** pp. 72-85 (August 1986).

[Wal64] Wallace, C. S., "A Suggestion for a Fast Multiplier," *IEEE Transactions on Electronic Computers* **EC-11** pp. 14-17 (February 1964).

[Wes85] Weste, N. H. E. and K. Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley Publishing Company, Reading, MA (1985).

VITA

Captain David M. Gallagher was born in Stillwater, Oklahoma on 7 February 1956. Following graduation from high school at Alamogordo, New Mexico in 1974, he received an appointment to attend the US Air Force Academy at Colorado Springs. He graduated from the Academy in May 1978, with a degree of Bachelor of Science in Electrical Engineering and a commission in the USAF. After completion of Undergraduate Pilot Training, Captain Gallagher was assigned to TAC and has flown the F-4E and F-4G Wild Weasel. He was assigned to the 37th TFW at George AFB, CA prior to entering the School of Engineering, Air Force Institute of Technology, in June 1986. Following graduation, Captain Gallagher will be assigned to AFOTEC/OA at Kirtland AFB, NM.

Permanent Address: Route 3, Box 591

Duncan, Oklahoma 73533

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for Public Release; Distribution Unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GE/ENG/87D-19	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, Ohio 45433-6583	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Air Force Off. of Scientific Research 8b. OFFICE SYMBOL (If applicable) AFOSR/XP	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) Bolling AFB, Washington D.C. 20332	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)
RAPID PROTOTYPING OF APPLICATION SPECIFIC PROCESSORS (Unclassified)

12. PERSONAL AUTHOR(S)
David M. Gallagher, B.S., Captain, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS Thesis	FROM _____ TO _____	1987 December	182

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Computer Architecture, RISC, Integrated Circuits, Microprocessor, VLSI, Computer Aided Design
09	01		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Chairman: Richard W. Linderman, PhD., Captain, USAF
Assistant Professor of Electrical Engineering

Approved for public release: IAW AFR 190-1.
31 Dec 87
Dean for Research and Professional Development
Air Force Institute of Technology (AFIT)
Wright-Patterson AFB OH 45433

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Richard W. Linderman, PhD., Capt, USAF	22b. TELEPHONE (Include Area Code) 513-255-3576 / 22c. OFFICE SYMBOL AFIT/ENG

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. ABSTRACT

Numerous applications throughout the Department of Defense, industry, and academia require the design of custom processor architectures. Design of these processors, however, is normally a lengthy process. This thesis defines a methodology for rapid prototyping custom VLSI processor architectures. Using this methodology, the design and implementation of application specific processors can be reduced from several years to two months. This reduction makes a high-performance VLSI solution feasible for Department of Defense applications that would otherwise have settled for a lower-performance alternative.

The rapid prototyping methodology is based upon the specification of a general purpose architecture customized via microcode to solve unique applications. Since processing requirements will vary, the designer chooses appropriate macrocells from a design library to provide the best hardware support. A high-level language description of the problem is then translated into microcode. The microcode is automatically assembled and designed into a ROM (read-only memory), resulting in a processor customized to solve the given application. By allowing the designer to quickly convert ideas into implementations, the rapid prototyping methodology frees the designer to be creative rather than becoming mired in implementation details.

A general purpose VLSI architecture was designed to support the rapid prototyping methodology. The control section of the architecture centers on the microcode ROM (read-only memory) and a microcode sequencer, which provides proper addressing to the ROM. The datapath section (I/O path, registers, and arithmetic hardware) uses the control signals from the ROM to perform the required processing. The datapath macrocells were designed in a "bit-slice" fashion, allowing easy configuration to different data types and widths. A prototype processor was implemented to test the architecture for functionality, performance, and operating characteristics. Additionally, a parallel floating point multiplier, applying Booth's modified algorithm in a Wallace tree structure, was fabricated. To further support the rapid prototyping methodology, several design tools were developed. These include a program to automatically generate a ROM in the Magic format and an extraction tool which generates a VHDL description of a circuit from a transistor listing, allowing high-level simulation of the circuit and thereby closing the "design loop."

The rapid prototyping methodology has been successfully applied to three different applications. These applications demonstrate that a custom application specific processor can be designed in less than two months using this methodology. This reduced design time also translates into reduced cost and program "risk." This dramatic decrease in design time could result in a significant increase in the usage of VLSI/VHSIC solutions to Department of Defense applications.

END

DATE

FILMED

MARCH

1988

DTIC